Development of a cluster of LXC containers

A Master's Thesis
Submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by
Sonia Rivas Quintana

In partial fulfilment
of the requirements for the degree in
MASTER IN TELECOMMUNICATIONS ENGINEERING

Advisor: Jose Luis Muñoz Tapia

Barcelona, October 2017

Contents

Chapter 1

Introduction

Abstract

The consensus algorithms were designed to reach agreements in distributed systems trying to maintain a certain tolerance to failures. An application derived from these algorithms is ETCD, a type of key-value storage that provides a safe and reliable way to save data shared in a network of servers, allowing the preservation of the information in spite of the possible fall of the involved nodes .

My thesis shows the design and operation of these algorithms, and addresses the question of how to manage the ETCD database and which features involves, as well as recommendations of configuration settings to achieve the best performance of the system.

Finally, an academic practice is presented, aimed to network students that would want to have a deeper knowledge about consensus algorithms applications in a distributed system.

Acknowledgments

I would like to express my gratitude to my supervisor, José Luis Muñóz, for his support and help during the development of this project, solving the doubts that were emerging and giving me clues to implement his suggestions, which have allowed me to investigate other aspects related to the main theme of this document.

Furthermore I would like to thank Nabil El Alami for his help in shaping this document, since I have used his knowledge about the software used, LaTEX, in order to achieve the desired aesthetic.

Last, but not least, I want to thank my family for all the support received during the completion of the master's degree at the UPC, which has allowed me to finish it with the same motivation and enthusiasm as when I entered.

To all of you, I thank you very much.

Revision history and approval record

Revision	Date	Purpose
0	15/05/2017	Document creation
1	10/10/2017	Document revision
2	13/10/2017	Document approval

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Sonia Rivas Quintana	soniarivasquin@gmail.com
José Luis Muñoz Tapia	jose.munoz@entel.upc.edu

Written by:	Reviewed and approved by:
Date: 10/10/2017	Date: 13/10/2017
Name: Sonia Rivas Quintana	Name: José Luis Muñoz Tapia
Position: Project Author	Position: Project Supervisor

List of Figures

List of Tables

1.1 Introduction

The main goal of this section is to provide a summary of the actual project. All the extended documentation of the researches of this project and a academic practice are contained from the following chapter of this document.

1.1.1 Statement of purpose (objectives)

The purpose of this project is to analyze and deploy a cluster of LXC containers. Several Linux containers (LXC) would be used, because they provide a non-package manager as a way for distributing payload applications, requiring instead all applications to run inside their containers. Therefore, the programming environment will be Linux, being bash the programming language used. This system would contain a shared database called ETCD, which would help to manage the information among the cluster nodes based on consensus practices.

Three goals were establish at the beginning of the project:

1. **Consensus algorythms research**: this would be the background of the project, due to it explains how the algorithms on which this system is based work.

2. **ETCD research**: documentation of this facility to understand its operation.

3. **ETCD practice**: practice with academic aims about ETCD.

1.1.2 Methods and procedures

The steps followed in this project could be divided in two:

- **Research**: first of all, it was required to perform a deep research about the consensus algorythms in order to give a background to the project. Next, it was made the same procedure with ETCD, as the main topic of this document.

- **Practice**: after achieving a complete view of the environment, a practical part based on explanations, commands suggested and questions was developed in order to reinforce the knowledge acquired in the previous step.

1.1.3 Requirements and specifications

- Conceptual requirements:

This project requires some experience in working with Linux, since the operations carried out during its development have been implemented in this environment. It is also necessary to know the operation of a network of nodes, because we work with a cluster of servers that have in common a resource to share information. Certain basic knowledges about OSI model, like network or application layers, are also useful, because some protocols as IP and HTTP are used repeatably.

- Technical requirements:

As this project has been developed in Linux, it is necessary a computer with a Ubuntu SO installed, as well as enough memory to host several Linux Containers.

1.1.4 Third party resources

For this project, the following tools have been used:

- Ubuntu.
- Linux Containers (LXC).
- ETCD v. 2.2.5
- Linux command-lines: cURL and etcdctl.

1.1.5 Work Plan

Work packages:

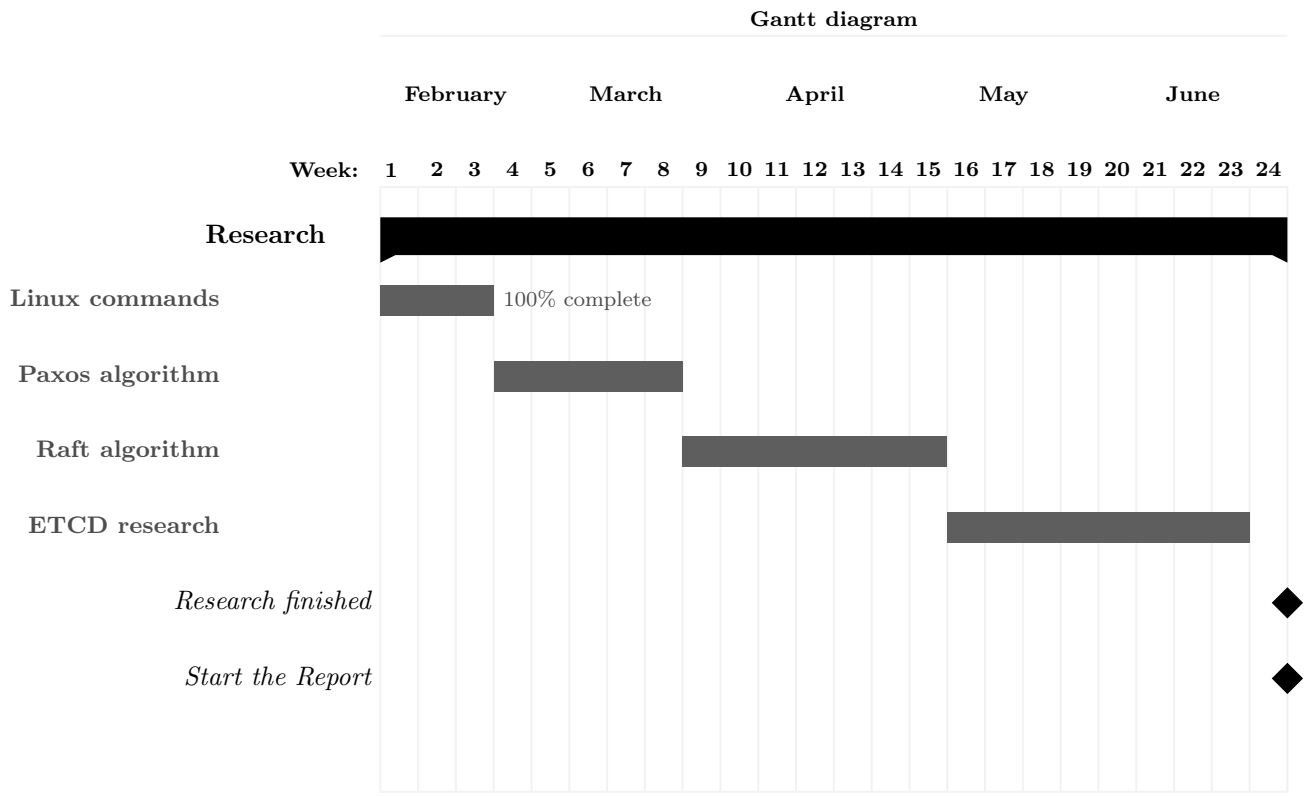
Project: Linux command research	WP ref: WP1	
Major constituent: Linux	Sheet 1 of 4	
Short description: Review of all Linux command that could be involved in the Project development.	Planned start date: February Planned end date: February	
	Start event: February End event: February	
Task: Linux command review	Deliverables:	Dates: Weeks 1-3

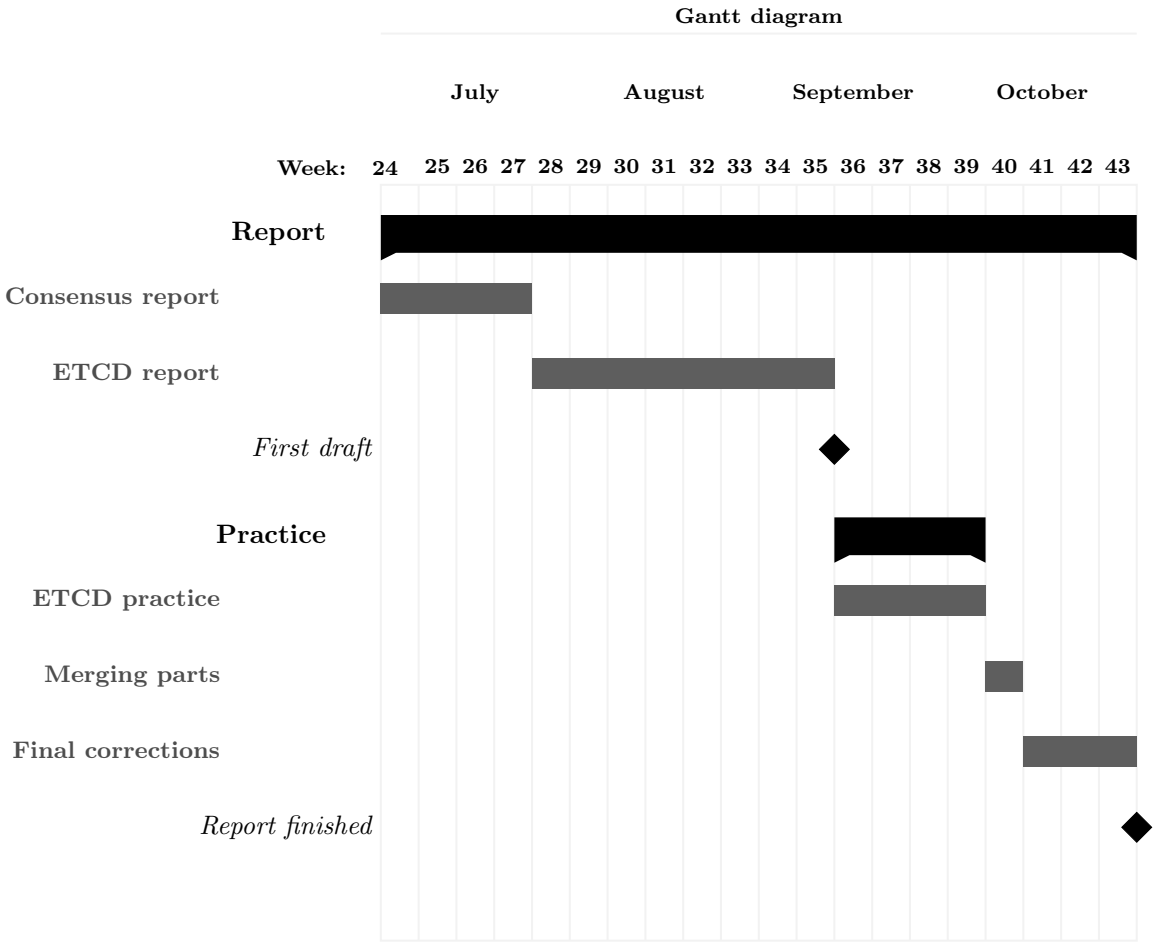
Project: Project research	WP ref: WP2	
Major constituent: consensus algorithms, ETCD.	Sheet 2 of 4	
Short description: Deep research about the main topics of this project.	Planned start date: February Planned end date: June	
	Start event: February End event: June	
Tasks: 1. Paxos algorithm research. 2. Raft algorithm research 3. ETCD research.	Deliverables: Documentation	Dates: Weeks 4-24

Project: ETCD practice	WP ref: WP3	
Major constituent: ETCD	Sheet 3 of 4	
Short description: Academic practice with commands, suggestions and questions about ETCD operation.	Planned start date: September Planned end date: September	
	Start event: September End event: September	
Tasks: ETCD practice.	Deliverables: Documentation	Dates: Weeks 35-40

Project: Report	WP ref: WP4	
Major constituent: ETCD	Sheet 4 of 4	
Short description: Report about the consensus and ETCD research and ETCD practice.	Planned start date: July Planned end date: October	
	Start event: July End event: October	
Tasks: 1. Consensus report. 2. ETCD report. 3. Practice report. 4. Merging and final report.	Deliverables: Documentation	Dates: Weeks 24-43

Gantt diagram:





1.1.6 Deviations from the initial plan

No.

Chapter 2

Consensus

2.1 Introduction

When a nodes system architecture is being designed, depending on the type of application to which it is intended, different decisions could be taken. On the one hand, a designer could like to implement a centralized model, where a single host executes the majority of system logic. The main advantage of this kind of architecture would be the capability of offering a more manageable and simple system software, besides robustness. On the other hand, it has an important disadvantage, which would be its fault tolerance: if the main host fails, the entire network will fail with it. For the purpose of this project, a robust fault tolerance is fundamental since in the case of a client making a request, the system could not allow errors when emitting an answer due to a node that was dropped.

The architecture of a distributed model is based on the cooperation of several nodes in order to provide a service. These nodes support various processes and/or databases with which they coordinate their activities and exchange information about the application, the network, or themselves. This type of structure provides scalability to the network, allowing its growth without affecting existing nodes. However, when relaying logic on several nodes, this type of system is prone to failures, either by a node that fails to send messages (dropped node) or arbitrary messages and erroneous messages (corrupt node).

The strategy to be followed by distributed systems is based on the replication of logs, which would act as a backup in case one of the nodes fell and the information that was stored was lost. Herein lies one of the pillars of this project: the consensus algorithms. In the case the network experiences errors that provoke the nodes to fail or send erroneous messages, the standing nodes should agree to offer the client a solution that is free of inconsistency.

Broadly speaking, the methodology followed by consensus algorithms is simple. Given a collection of nodes that communicate through messages between them, the goal is to reach consensus by assuming that communication is reliable, but that nodes can fail. The steps would be:

1. Each node ni begins the cycle in a non-decision state, proposing a value vi.
2. Nodes start to communicate by sharing their vi value.
3. Finally, each node sets the chosen value in a decision variable di, entering in a decision state that cannot be changed.

However, there is not a unique approach to consensus. In this way, we can find in the literature two differentiated types of consensus algorithms:

- **Symmetric**: this means, all the nodes have equal roles, without a leader handling client requests. An example of a symmetric consensus algorithm is Basic Paxos.

- **Asymmetric**: there is a node with a leader role, which permits it to be the one handling the client requests. The rest must accept its decision, with a passive role. This approach is more efficient, as avoids typical conflicts where every node has the same role. Examples of a asymmetric consensus algorithm are Multi-Paxos and Raft.

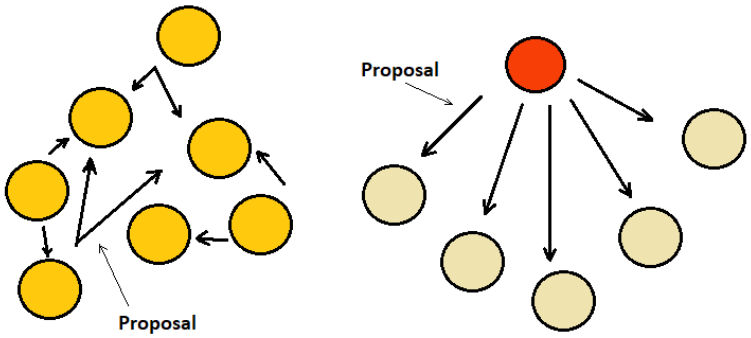


Figure 2.1: Cluster configurations: a) Symmetric b)Asymmetric

Regardless of the type of approach implemented, consensus algorithms must meet three conditions:

- **Consistency**: the value decided by all nodes must be the same.

- **Termination**: the nodes must end up assigning a value to their decision variable.

- **Integrity**: If all the correct nodes have proposed the same value, then any correct node in the decision state has chosen this value.

In the following sections, it will be described the operation of the main consensus algorithms, Paxos and Raft, in which ETCD is based. The first algorithm has in turn two variants, which would be the Basic Paxos and the Multi-Paxos. Raft is a later algorithm based on Paxos whose purpose is the same, but its conception is simpler. As these algorithms are only the background of the project, they will be shown in general terms, remarking the fundamental aspects related to ETCD.

2.2 Paxos Algorithm

Leslie Lamport proposed Paxos in 1998, describing the ancient Greek civilization on the Paxos island. Based on its political manner to take decisions, he tried to replicate it into distribution systems.

As explained in the introduction, the goal of a consensus algorithm is that each node has the same information to be able to respond properly to client requests. In addition, it is necessary that all of those nodes have the same State Machine, which would be the application that it is using the algorithm, because if it received the same commands executed by the clients in the same order, the logical procedure is that they would act in the same way. Thus, if one of these nodes fails, another can continue to provide the required service.

In short, the procedure would start when a client sends a command to a cluster node. It collects it in the "Consensus Module", responsible for the proper log replication, and proceeds to save it in a local log. Then, this node would send the same command to the rest of the cluster, which would replicate it in its respective local logs. Once everyone has saved the command, it would be passed to its State Machine for execution. When one of the machines has executed the command, the result is returned to the client. This type of system only works correctly if there is a quorum, that is, a majority of nodes are up and functioning correctly. The only faults it accepts are from dropped nodes or lost or delayed messages, but it does not contemplate corrupt nodes acting maliciously, which would be a Byzantine failure model.

There are two versions of the Paxos algorithm: Basic Paxos and the Multi-Paxos. The first one, as its name implies, is the simple approach: it consists of one or more nodes proposing values (which they have received from the client as a request) of which only one can be chosen at a time. Multi-Paxos, on the other hand, is a more complex approach, in which several values are grouped in a log at the same moment to save time eliminating unnecessary delays.

In the following subsections, it will be provided a general description of these variants in terms of its composition and functioning, as well as issues faced.

2.2.1 Basic Paxos

First of all, an overview of the simplest form of Paxos algorithm will be presented. When we talk about the cluster composition in Basic Paxos, two different agents can be differentiated:

- **Proposers**: nodes with an active role that have received the requests of the clients and, based on them, propose values to the rest of nodes with the aim to be chosen.
- **Acceptors**: nodes with a passive role whose objective is to answer the messages of the proposers, accept the values proposed and finally save in the State Machine those chosen by the cluster.

However, a series of rules must be followed to be able to execute this consensus algorithm correctly. At least two fundamental requirements must be fulfilled:

- **Safety**: property that guarantees that the algorithm will not do something wrong; only one value at a time will be chosen each time and a node will never store in its State Machine a value that has not been chosen by the cluster.
- **Liveness**: nodes must communicate reasonably fast and with most of the cluster running. If this is accomplished, the processes will end up choosing a value eventually, and this one will be stored.

This procedure was developed over time though. One of the first approaches was that of Srawman, in which a simple acceptor decided which values were chosen. The problem here was a situation in which the acceptor failed after having chosen the value. That is why the idea proposed was to have a quorum of acceptors, in which multiple nodes had to keep a value stored (this means, chosen) only if it was decided by the majority. In this way, if a node fails but quorum remains, it is possible to continue with the algorithm properly. However, this methodology caused a second problem: if the acceptor only chooses the first value that is sent to it, but simultaneously receives concurrent requests, no value would be chosen. Because of this, if a quorum is not achieved in a round, you must choose another in the next, until you get the majority (liveness property). Another point is that if the node must accept all the values that arrive to it, could choose multiple values, and this contradicts with the requirement of safety. The solution is that a proposer node, before suggesting another value, has to look at whether a value has been chosen by its network, and accept it as well. Old proposals are discarded in favour of new ones, and therefore an order must be kept in the RPCs so that old values are not accepted. At this point it is important to remark that "accepted" is not the same as "chosen", because a value is chosen when a quorum of nodes has accepted it. Then the chosen value is

passed to the State Machine to be executed. A node can accept a value, but if the quorum chooses another one, that value will not be stored.

As we have said before, among the proposals must be an order not to violate the property of safety, and therefore each of them has a unique identification number. As expected, the newer proposals have higher numbers because they are priorities, and it is the proposer who is responsible for the assignment of this number taking into account that it should never have been used (this includes the reuse of proposals numbers after a fail or restart of a server). When an acceptor receives several proposals, it should check which one has more priority to discard the remaining ones.

The table 2.1 shows the format to be followed by all proposals numbers, with two fields. As the last proposals have priority, the first field indicates the number of the round, and in the second, the identifier of the proposer (eg if node number 6 proposes a value in round number 10, the proposal will have the identifier 10.6).

Proposal number	
Round Number	Server ID

Table 2.1: Proposal format

After the explanation of the general composition, the procedure below Basic Paxos will now be described. Broadly, it consists of two phases:

1. **Preparing RPCs broadcast**: the node looks for any values chosen by the cluster and blocks old proposals that have not been completed. If it finds a value chosen by the quorum, it discards its own value and proposes the same value as well. If not, prepares acceptance RPCs with its initial value.

2. **Accept RPCs broadcast**: proposes to the nodes a specific value to be accepted by the quorum and finally chosen by the cluster.

Going through Basic Paxos in more detail, it is essential to note that the algorithm forces acceptors to have three variables stored on disk: "*minProposal*", "*acceptedProposal*" and "*acceptedValue*". In this way, the process would begin when a proposer chooses a new proposal number n, which should never have been used before and taking into account that n is the current round, that is, the highest priority. Then it broadcasts this packet *Prepare(n)* to all nodes. Receivers, also known as acceptors, look at the packet and respond: if n is greater than the value they had stored for the proposal number (stored in the "minProposal" variable), it returns an "*acceptedProposal*". If it already had an accepted value, it returns it also in the variable "*acceptedValue*". The proposer receives the response and replaces the "*acceptedValue*" with its current "*value*" if it has been notified by the acceptor. At this point, phase two of the algorithm begins: the proposer broadcasts the *Accept(n, value)* package to all acceptors, which have to respond again. If "n" is still the highest number (corresponding to the last round), they update their "*acceptedValue*" variable with the "value" received in the RPC. Also, they update their "*acceptedProposal*" and "*minProposal*" variables with the value of "n" and return "minProposal" to the proposer. It waits to receive response from most nodes to obtain a quorum, and in case the received value is equal to the initial n, that value will be chosen. If the answer consists a value greater than n means that there is another more priority proposal, and you should go back to the beginning.

The algorithm contemplates three scenarios when a new proposal is launched with another one in progress:

1. **A previous value has already been chosen**: with the *Prepare(n)* package, the last proposer realizes that a value has already been chosen by the quorum, and updates its own value to launch the acceptance

packet with the one already chosen by the majority of the cluster. For example, if node 1 proposes the value X and it is chosen by the quorum, although node 2 subsequently wants to propose the value Y, it must abandon it to propose the value X as well.

2. **A previous value has not been chosen, but the new proposer sees it**: even if there is no quorum at the moment, if the new proposer sees a node sending an acceptance packet it must assume that the rest will also accept it, so it will discard its new value and would adopt the old as in the previous case.

3. **The previous value has not been chosen, and the new proposer does not see it**: even if there is a node that has accepted an old value, if the new proposer does not see it, it will launch its acceptance packet with its own value. So, even if an overlapping node sees the two acceptance RPCs, it will stick with the latter because it has more priority. In this case, node 2 would triumph and the chosen value would be Y.

However, this algorithm is not perfect. If new values are proposed before the packages are accepted, the latter will have priority, and if it happens several times you can fail into an endless loop where the acceptors cannot finish choosing a specific value (like a continuous case 3, decribed previously). This means, with multiple proposers working at the same time, there are more conflicts due to the high load. A proposed solution was to introduce a random delay before returning to make a proposal, so that it is given the opportunity to the proposer to end its cycle. But this is not an optimal solution, as it implies a delay added.

An additional factor to keep in mind is that Basic Paxos is inefficient also at a resources level, since it is compulsory to use two rounds of RPCs (Prepare and Accept) to choose a single value. To solve these questions, a more complex algorithm was sought that started from the same initial idea, and thus was conceived Multi-Paxos.

2.2.2 Multi-Paxos

Multipaxos is a generalization of Basic Paxos, developed to face the problems that it could not solve efficiently. It was sought a way to optimize the operation of the algorithm, ensuring a complete replication of the stored logs and contemplating different changes in the cluster configuration, as well as save more resources inncesarialy wasted. Nevertheless, at the beginning we have to remark that, while Multi-Paxos has been extensively implemented in many systems, it has not been specifically explained in the literature, detailing how it solves these issues. This, together with the great complexity that the algorithm entails, leads to the present project introduction being shown in the most general and simplest way possible.

The procedure to follow would be roughly the same as Basic Paxos: a client sends a command to the server to be executed, the server uses Paxos as a consensus algorithm to propose the value associated to be chosen by the cluster and then, after entering it into the State Machine, it returns a response to the client. The main variation is that a single command does not occupy the whole log, but only one entry of the same. In this way, several commands can be accepted in a single round, unlike in Basic Paxos, that could only be chosen one (safety property). Basic Paxos is executed in each log entry, adding extra arguments to the "Prepare" and "Accept" request that will be explaned later. The acceptor waits for the previous log entries to be applied and then executes the new command in the State Machine, returning the result to the client who requested it at first.

As mentioned before, there are some issues that Multi-Paxos need to remedy:

1. **Optimize the performance of Paxos**: reduce the conflicts that occurred in Basic Paxos between proposers, by introducing a single leader, who would act as the only proposer, and eliminate a large part of Prepare requests to save resources. In this sense, you would only need one Prepare for the entire log.

2. **Ensure complete replication of logs**: in Basic Paxos only the proposer knew which commands had been chosen, not the entire cluster.

3. **Manage the way customers have to deal with a server crash.**

4. **Add and remove nodes to the cluster without affecting the service.**

Emphasizing the Multi-Paxos algorithm, the immediate question that may arise is what is the most appropriate way to choose the log entry for a particular client request. Returning to the process that follows, the client would send to a node a command to be executed. This server would then look for the first log entry that did not have a command already chosen, and would start Basic Paxos in the way explained above for this particular index. If this index is already working with a previous value, it would discard its own and it would start again. If not, it would propose the new client command. Thanks to this procedure, servers can handle multiple client requests by selecting different log entries for each. They must then be executed in the State Machine in the correct order.

Returning to the optimization of the Paxos algorithm as far as its performance is concerned, there has been several thinking about replacing multiple proposers with a single one, which would act as a leader. There are many ways to choose this node, but we will explain here the simplest, conceived by Leslie Lamport. In this case, the server with the highest ID would be chosen as the leader, and it would send a heartbeat to the rest of the cluster every T ms to record that it is working correctly. In case a node did not receive this heartbeat in at least 2T ms, it would act like a new leader that would propose to the rest of the cluster the new requests of clients commands. The rest of the nodes should reject these requests and send them to the leader, because they can only act as acceptors, not proposers.

Because there is only a single proposer in the cluster, another point to improve process performance and avoid spending as many resources as possible is to reduce Prepare packets. These, as we have mentioned, are intended to ascertain whether the node has a value currently in the process of being accepted, and if not, to propose its own value. It blocks so old proposals, now indicating proposal numbers for the entire log, not just an entry. With this single Prepare, it finds out if there are values chosen and return the proposal with the highest ID for the current entry, other than the variable "*NoMoreAccepted*", which indicates that there are no accepted proposals beyond the current entry. In case the proposer receives this variable, will avoid sending more packages Prepare because it already knows its situation. The leader will resume Prepare RPC transmission when an Accepted packet is rejected.

However, there is one really important point that Multi-Paxos must solve, and that is to get the complete replication of the logs on all the nodes of the cluster. So far, only the majority, the quorum, had the correct logs replicated. Likewise, only the proposer knew which values had been chosen. The solution to get a full replication and that all nodes have the same information, achieved in four phases:

1. The proposer sends Accept RPCs packets until the acceptors respond, in the background. Thanks to this method, much of the logs are replicated correctly. However, it does not solve that all servers know which entries have been chosen.

2. Nodes mark the entries as chosen by matching the "*accepterProposal*" variable of that index to infinity. In this way, the proposal will never be overwritten because another one will not be received with a higher value ID. Also, each server will have a variable called "*firstUnchosenIndex*" that will indicate the first entry that has not been marked as chosen. This will show that all the previous ones have already been chosen.

3. The proposer communicates to the acceptors the entries chosen using the variable "*firstUnchosenIndex*". The acceptor will mark all entries as chosen if they are less than this variable and if the variable "*acceptedProposal*" is equal to "*request.proposal*" for that particular index. With this method, the acceptor node will know almost all the chosen entries.

4. The acceptor returns its "*firstUnchosenIndex*" variable in response to Accept, and if this variable in the proposer is larger than in the acceptor, the proposer sends a Success package in the background. This package indicates for a certain index the value of the "acceptedValue" and marks the "*acceptedProposal*" as infinite so that it is not overwritten. In addition, it returns its "firstUnchosenIndex" to update the Acceptor.

This explanation would simplify the operation of the Multi-Paxos algorithm in general terms. However, two issues remain to be resolved: what happens if a node fails after receiving a request from a client? And how does the algorithm face a node failing? Would be the service stopped? Let's review the way the customer communicates with the cluster. A command is sent to one of the nodes to return a result. In case this node is not the leader, it is forwarded to that server. If it is, the server runs Multi-Paxos. In this scenario, we can face two types of problems:

1. **The leader fails just after receiving the request**: the command will not be executed, because the rest of the cluster has not received the proposal. For this reason, a request timeout is established, whereby if the leader does not respond at a time T to the client, the client retry the request with another node, which will eventually redirect the command to the new leader.

2. **The leader fails just after executing the command, but before responding to the client**: in this case, the command should not be executed twice, so the server must include the id of each command in the log entry and then save it to the State Machine next to the most recently executed commands. In case the request timeout expires, another leader will take over and re-run Multi-Paxos. However, before running it, it will look at the State Machine to see if the command has already been executed. If so, it will ignore the new command and return the response that the old one had generated.

The last point to be addressed by this introduction to Paxos is the way in which the cluster manages the configuration changes. For example, it grows from 3 nodes to 5, or a node is dropped because of a technical fault. Consensus algorithms must support these changes that are critical to their proper functioning, because the quorum changes in several cases. If we initially have a cluster composed of 3 nodes, the quorum would be 2; if it grows to 5, the quorum is 3. What would happen if two configurations coexist at the same time, the old and the new, choosing both majorities different values for the same log entry? In such a case, the safety property would be violated. In addition, the dropped node has to be replaced and the degree of replication changed.

Leslie Lamport proposed the following solution: to use the same log filled with client commands to indicate configuration changes. In this way, the configuration is saved as a log entry that is replicated as such, communicating the change to all nodes. Nevertheless, concurrency must be limited to avoid errors, and this is achieved by the Alpha parameter. This variable indicates how many new entries will be applied later, not the immediate change. The following example in Figure 2.2 shows what would happen when the configuration changes twice considering that alpha is set to 4.

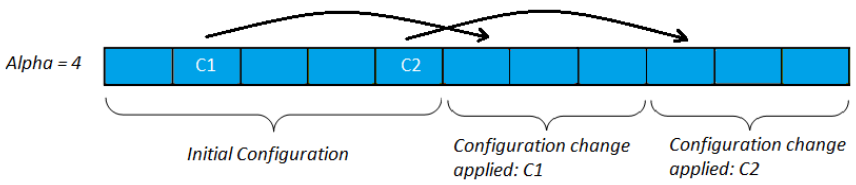


Figure 2.2: Paxos configuration changes procedure

As has been said at the beginning, although this algorithm works, its complexity makes it difficult to understand and apply. Likewise, there are certain problems that do not completely resolve, and the proposed solutions are not clear. For this reason, it was decided to create a new consensus algorithm based on Multi-Paxos, which simplified its design and was understood more intuitively, but with the same efficiency in its operation.

2.3 Raft Algorithm

Designed as an alternative to the Paxos complexity by the Stanford University laboratory, Raft is another consensus algorithm for managing log replication in a cluster of nodes. While in the Paxos algorithm the problems are mixed in a confusing manner and solved in the same way, Raft tried to separate each subject clearly differentiated in order to solve it in phases. However, the general operation of the algorithm is equal to Multi-Paxos. Again, a client executes a command and a cluster node receives it, passing it to its Consensus Module. If this node meets some requirements, it will pass it to other nodes to accept and choose, replicating it in an entry of its log and passing it to its State Machine for execution. At that point, the initial node returns the result that has been generated to the client. We remember that this model of failures only accepts fails or delays in the nodes, but not that there is a malicious one that tries to harm the correct operation of the cluster. In that case, Raft, like Paxos, would fail.

In this variant, a new way of dividing the nodes of the cluster was included. The diagram in Figure 2.3 shows the three possible states of a server, which would be:

1. **Leader**, which would handle all client requests and the correct replication of the logs. In each round, this server must notify the other nodes that is up and properly operating by means of RPCs, and in case a timeout is exceeded without a response, the cluster would enter a new election period. There can only be one leader in the cluster to avoid conflicts (as it happened with Basic Paxos).

2. **Follower**, a passive node whose sole purpose is to respond the requests it receives from leaders and candidates for leader. In case of a crashed leader node, it becomes a candidate.

3. **Candidate**, nodes who have previously been followers and choose to become cluster leaders. If the majority vote them, they increase their rank; if they see that another has been chosen, they return to be followers.

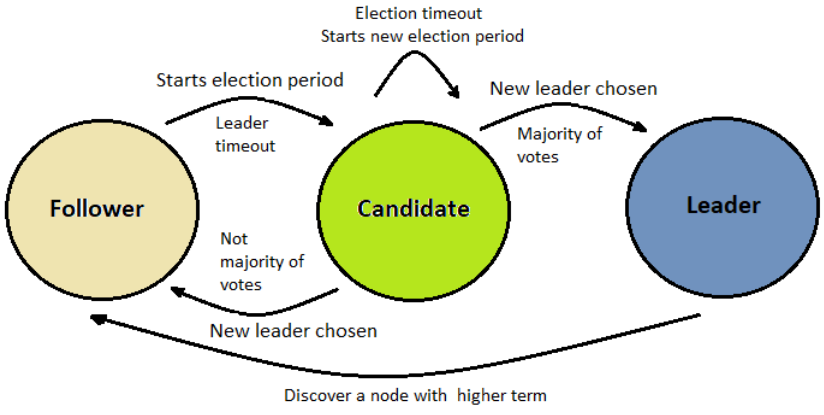


Figure 2.3: Nodes roles cycle

In this section, we will break down Raft's algorithm into its key points: how it performs log replication, choice of leaders, the client protocol, procedure followed for cluster configuration changes and how it is achieved to remain secure and consistent algorithm in spite of these changes. For this it is important to point out that the time in which the algorithm works is divided in terms (Figure 2.4). Each term usually has two phases: an initial one in which a new leader of the cluster is chosen, and the normal operation, in which the algorithm works under the command of a single leader. As you can guess, the end of a term means that the leader node has failed, and at that moment the followers become candidates to choose a new one. However, as can be seen in the figure, in term 5 this election does not always come to the desired end. If the election timeout is reached and no new leader has been chosen, a new term has to be started with a new election, until one is finally picked, as would be the case in term 6. Transitions between states can be seen in Figure 2.3.

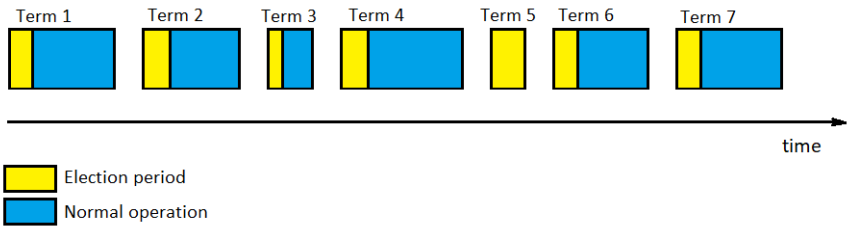


Figure 2.4: Term phases

This system of terms aims to find obsolete information, logs that it is not convenient to replicate. In the hypothetical case that the leader of term 3 is temporarily out of service, it would not be able to send the corresponding heartbeat to the rest of the nodes. This would lead to new elections and the transition to a new term, the fourth. What would happen if the term 3 leader returned to work correctly and sent its logs to the rest of the cluster for its replication? If the nodes accepted it, the consistency would be lost, since they would surely already have new log entries for this term 4. That is why leaders send their replication request RPCs in the "*AppendEntries*" function. Among its arguments, we highlight two: the **identification of the leader** and the **term in which he is**. The followers would see that node is in a previous term, so they would ignore their request, and the new leader would send a RPC to that obsolete leader indicating that the term has changed, which would lead to degrade his status from leader to follower.

These "heartbeats", sent by the leaders to the cluster to indicate they are working and thus maintaining their authority, are actually RPC of empty *AppendEntries*, and the period in which they are sent can be modified. During the set up, it must be taken into account that the timeout is usually between 100-500ms for correct operation, so the heartbeat period should be much lower to cover possible delays in communication between servers. In case of not receiving this heartbeat, the cluster enters in election period. Each node increments the current term and changes its status to Candidate. They then vote to themselves and send "*RequestVote*" RPCs to the other servers to get their votes and become the new leader. It will not stop until one of the following cases occurs:

1. **Most of the cluster votes for him and it becomes the new leader.** At that moment, the node begins to handle the requests of the clients and sends "*AppendEntries*" to the rest of servers so much to maintain its authority as to request the replicate of logs.

2. **Receive an RPC from a valid leader (current term).** The node changes its status from Candidate to Follower and acts again passively.

3. **Do not receive the majority of votes neither a valid RPC of another leader, and the**

timeout is exhausted. The node remains as Candidate, increases the current term and restarts a new election period.

As it happens with the logs replication, two fundamental requirements must be fulfilled with the election of a new leader. The first one is **safety**, which indicates that there will be at most one leader per term since otherwise conflicts and inconsistencies will occur when replicating logs in the rest of the cluster. For this, in the election only a single node must be voted, and it must be ensured that two nodes do not get the majority of votes at the same time (that's why it is recommended to have an odd number of nodes in the cluster). The second requirement is **liveness**, which would ensure that a candidate eventually became a leader by winning the election. To achieve this, a random timeout is established between [T, 2T] taking into consideration that T must be a number much higher than the propagation time of the RPCs. In this way, the cluster ensures that the choice is made safely and does not end up creating chaos in the configuration.

Once overcome the correct choice of leaders, we enter in the normal operation phase. A client sends a command to one node to be executed, and that server returns the result. This node, if it is the leader, will send it to the rest of the cluster to be replicated; if it is a follower, it will redirect the request to the leader, who will be in charge of the operation. The sent RPCs, as mentioned above, are called "*AppendEntries*", and are used both for log replication and for the discovery of possible inconsistencies in the followers. The arguments included are as follows:

- *term*: indicates its current term, essential for the follower to know if that is the current leader or one obsolete from a previous term.
- *leaderId*: is the identification of the leader node, and is used so that the followers can redirect to the leader the requests of clients that arrive to them.
- *prevLogIndex*: index of the log entry that immediately precedes the new ones.
- *prevLogTerm*: term of the *prevLogIndex* entry.
- *entries[]*: log entries to be saved. In case it is a heartbeat, this argument is sent empty.
- *commitIndex*: last entry that is known to have been committed.

On the other hand, the follower returns two results:

- *term*: its current term, necessary for the leader to be updated.
- *success*: is "True" if the follower fulfills the entry matching given the *prevLogIndex* and *prevLogTerm*.

Once the command log entry sent by the client has been committed, the leader passes it to its State Machine for execution and returns the result. In addition, it notifies the followers that this entry has been committed by the RPC of "*AppendEntries*", and the nodes would then pass the command to their State Machines.

The structure of the log is similar to that one used in Paxos. In this case, the log is divided into entries that will keep each client command that is committed, and have three identifying parameters: **index**, **term** and **command** (Figure 2.5). However, in most cases the latter is not necessary because, if the input is committed, having the term and index would indicate the associated command.

These logs should be stored on a stable storage disk to overcome possible crashes when committed, that is, when they have been stored on most nodes in the cluster, after the command execution by the current leader. This action is optimal in the most general case, since the leader retries the sending of RPCs until

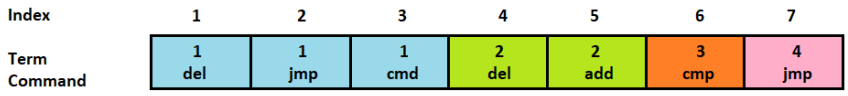


Figure 2.5: Log structure

most nodes respond satisfactorily. Moreover, it is verified that the logs are coherent with each other: all nodes for the same index and same term must save the same command, both in the current and previous log entry. This means that if an input is committed on a node, it implies that all the previous ones are also committed and are consistent with the majority of the cluster. The leader's way of checking if this consistency is fulfilled is by executing a matching entry using an induction step: as each *AppendEntries* RPC has the index (*prevLogIndex*) and the log term (*prevLogTerm*) preceding the current one, the follower should check that it matches its last log entry. If it matches, the node replicates the new command correctly; if not, rejects the request and the leader should be in charge of "purging" the wrong entries, as will be explained later.

Now than we know and understand the operation of Raft, it's time to explain why these "strange log entries" can appear in the cluster nodes. The common reason would be a change of leader: if the obsolete leader left partially replicated entries before failing, the nodes would not be up to date for the new term. However, the procedure to follow would be simple. The new leader would act normally, sending RPCs of heartbeats and requests for replication, and seeing that the logs do not match their own. In the next few iterations he would erase the erroneous ones and replicate those that have been commited. The more crashes of leader the cluster experiments, the greater number of erroneous log entries will appear. That is why it is necessary to reapply the requirement of safety, where once a log entry has been committed and therefore applied to the State Machine, that command cannot be overwritten in any way. This is due to a previous leader had already decided that entry had been committed, so it is correct and must be present in all the logs of the future leaders of the cluster. This ensures safety in three ways:

- Leaders will not overwrite entries in their logs because they are committed.
- Only entries in the leader log can be committed.
- Entries must be committed before being applied to the State Machine, at which point they are passed to the followers to do the same.

This requirement for algorithm security directly influences the choice of the new cluster leader. While in Multi-Paxos we presented a simple way to choose a new leader, using the ID of each node, this time we look for the best possible leader, since not all nodes are equally good for this task. Ideally, this server has the most complete log, that is, the largest number of commited entries. However, how can we say that an entry is committed? If a node is unavailable during the leader election, it may not be known which entries have been committed since they are not present on most nodes in the cluster (apparently, because maybe with the inclusion of those nodes temporarily unavailable, the quorum is reached). It is therefore a question of choosing the candidate with the log most likely to contain the largest number of log entries committed. To do this, candidates must submit a "*RequestVote*" RPC with the following arguments:

1. *candidateId*: identification of the node requesting the vote.
2. *term*: term in which the candidate is.
3. *lastLogIndex*: index of the last log entry of the candidate.
4. *lastLogTerm*: term of the last entry of the candidate.

The results returned by the requested node are:

1. *term*: current term for the candidate to be updated if necessary.
2. *voteGranted*: cast its vote. If it contains the value "True", the node has given its vote to the candidate.

If the term of the requested node is greater than that of the candidate, the first one denies the vote, because the other server would be outdated. The same would happen if they were in the same term, but the index of the last log entry of the requested one was greater than the candidate's index (the candidate would be in the correct term, but it does not have the complete log). This process is repeated throughout the whole cluster, and the candidate with the most likely to be committed log becomes the new leader if the majority decides so.

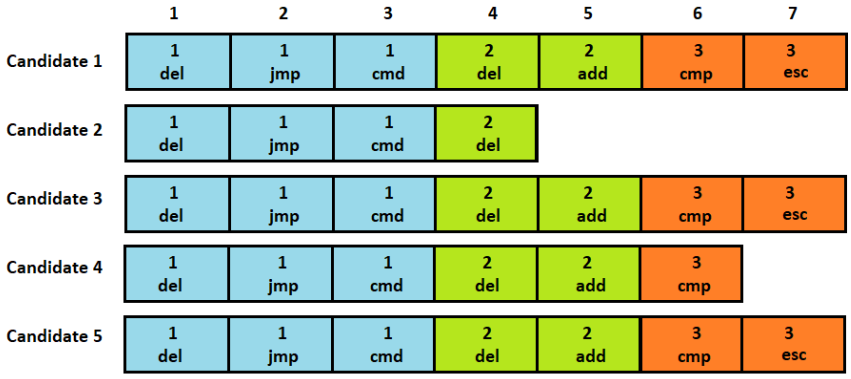


Figure 2.6: Example of election period

In Figure 2.5 it is possible to observe a cluster of 5 nodes in period of elections. After submitting their *RequestVotes*, candidates 2 and 4 would be out of consideration. But, for what reason? On the one hand, candidate 2 is not in the correct term, since it continues in the second, and still does not possess the complete log of that term. On the other hand, candidate 4 is in the right term, but does not have the last log entry that is assumed committed since most of the cluster contains it: index 7. Therefore, the new leader decision would be between nodes 1, 3 and 5.

A critical case that we can find is presented in Figure 2.6. In this situation, we are in term 4. The leader checks that the follower 2 does not have replicated index 4 of term 2, so he proceeds to send it to him. However, this entry is not committed securely. As it can be observed, the follower 4 has the most complete log, which indicates that it has been the leader of term 3. We can see that something must have gone wrong previously, because it does not have the log entries with index 3 and 4 of the term 2, and that node failed before being able to send the "*AppendEntries*" to the rest of the cluster nodes, so it is the only server with log entries of that term 3. In case the leader of term 4 failed, that follower could be chosen due to it is the node with the most complete log, even though this means that it will try to purge the log entries of term 2 when it became the new leader.

For this reason, new rules were added to achieve commitment in the log entries. For a leader, deciding that a commited entry, is not enough that it is stored on most servers, but also an entry of the current term must be stored in the same way in most of the cluster. This means that the last committed entry has to be the current term, due to this would "secure" those of previous terms. Thus, the follower 4 could not overwrite entries 3 and 4 of the log, since they have behind index entry 5 with a term more updated than

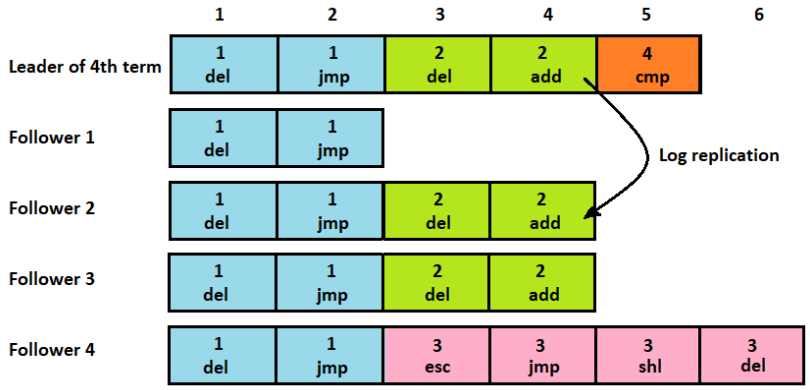


Figure 2.7: Logs replication problematic

theirs. In this situation, the leader of term 4 would purge the strange entries of that follower and replicate his log in the same one.

Previously it has been mentioned that the leader is responsible for repairing the logs of their followers in case of a mismatch with their own log. This has the aim of erasing the strange entries and filling in those that are empty with those that have been committed by the majority. The process would consist of the following: the leader maintains a variable called "*NextIndex*" for each follower, which as its name indicates, contains the index of the next log entry that will be sent to that node (initialized to 1 + leader's last index). If the matching fails after sending the RPC of *AppendEntries* and it is shown that there is an inconsistency, it decrements a unit the *NextIndex* and resends it until matching occurs. For example, if the leader has discovered that, of 8 log entries filled in by the follower, from entry 4 to the end are wrong, it will erase all log entries from that index. The next step the leader would take is to replicate from there the correct log, that is, the log entries committed.

Another point to be addressed by the consensus algorithms is the client protocol, that is, how clients act when the nodes they have contacted to request the execution of a command fail. Paxos and Raft work in the same way in this case, but we will review the performance of the algorithm again.

In Raft, a client contacts the cluster leader to send a command. In case he did not know which server is the leader, it is sent to any node, which would redirect him to the current leader of that term. There is a timeout, which once exhausted without response from the node that it trying to contact, the client would start its request again with another node in the cluster. Then the consensus algorithm would be executed, where the leader will not return the command to the client until it has been committed and executed by its own State Machine. But what would happen if after executing the command, the leader fails and does not return the result to the client? The client again waits until timeout expires, and once exceeded, retries the process with the new leader of the cluster. It is important in order to save resources that this command do not run twice. But in the practice, this would not happen, as future leaders have the old logs as long as they have been committed (and if executed by the State Machine, this is true). So, the new leader, before accepting the new command, would check his id to see if there is a matching log entry. If so, it would ignore the new command and return the result directly to the client, so that it was executed only once.

The last point that is going to be discussed about Raft is related to configuration changes (replacing a dropped machine or adding new nodes for the current cluster) procedure, that differs slightly with that one implemented in Multi-Paxos. As in that algorithm already commented, it is not possible to change directly from one configuration to another because not all the servers would change at the same time (each one will have a different delay). This situation would lead to a majority conflict, which would result in the choice of

more than one command by different quorums. A two-phase process is required to safely transition from the old configuration to the new one.

With this new perspective, many ideas were broadly discussed and at the end the Paxos algorithm was improved. If we remember, Multi-Paxos solution was to apply a delay, called alpha, that changed the configuration of the cluster in a time measured in log entries. On the contrary, the idea of a "joint consensus" was conceived in Raft, in which an intermediate phase encourages the creation of a transition membership between the old and new configurations. The change is applied in the form of a log entry, and is executed immediately after being received by the follower without having to be committed. Thus, during the intermediate phase of joint consensus, both majorities (old and new configuration) must agree to make the commands committed. Once this joint log entry is committed by the quorum, the leader sends the new configuration to the cluster. Again, the configuration is applied immediately upon receipt, and at that time a single majority will decide on the cluster, successfully completing the change.

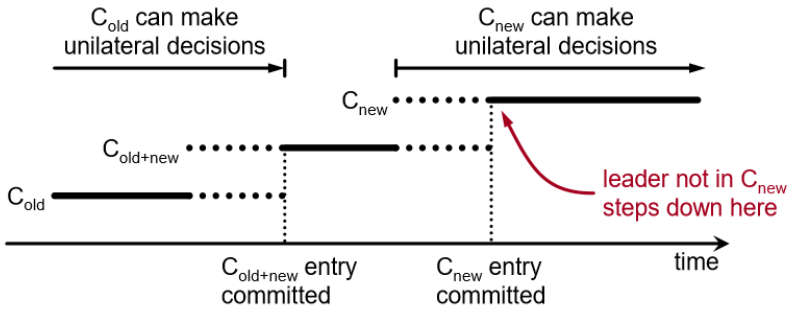


Figure 2.8: Raft configuration changes procedure. Source in [3]

The term leader can be found in both the old and new cluster configuration. However, if it is in the old and not in the new one, once the new configuration is committed, that server will lose its rank. A new choice of leader would occur, entering a new term and moving on to the subsequent normal operation.

In this way, we would conclude with the introduction of consensus algorithms, after describing how the elements of the cluster would communicate with each other and how they would cope with possible technical problems, such as delays, node failures and logging inconsistencies (again, it is important to keep in mind that its fault tolerance model does not contemplate the possibility that some node behaves maliciously). As the reader must have noticed, Raft is a consensus algorithm much simpler to understand than Paxos, and solves its problems much more intuitively and applicable to modern distributed systems. Therefore, from complex applications to the simplest, such as an online game, this algorithm is implemented to keep communication reliable and stable.

From here, we will focus on the main axis of this work, which would be ETCD. It is a secure and reliable form of information storage, which based on Raft proposes a support for applications to read and write in its database by means of key-value pairs.

Chapter 3

ETCD

3.1 Overview

The name "ETCD" was originated from two ideas: the unix "/etc" folder and "d"istibuted systems. This is due to the "/etc" folder is a place to store configuration data for a single system, whereas ETCD stores configuration information for large scale distributed systems. In other words, a "d"istributed "/etc" is "ETCD". This project is in continuous development, with a community working on it using the GitHub platform to share and implement new ideas or to solve critical issues, always trying to maintain the ETCD as simple as possible.

ETCD can be more technically described as a daemon that runs across all computers in a cluster, providing a dynamic configuration registry. This kind of setting allows various configuration data to be easily and reliably shared between the cluster members. All changes in stored data are reflected across the entire cluster because, as said before, the key–value data stored within ETCD is automatically distributed and replicated with automated master election and consensus establishment using the Raft algorithm. On the other hand, the achieved redundancy prevents failures of single cluster members from causing data loss. Moreover, ETCD is written in Go, which has excellent cross-platform support and small binaries.

However, latency in ETCD nodes is the most important metric to track, due to the fact that a severe latency would introduce instability within the cluster because Raft is only as fast as the slowest node in the majority. The best practice to avoid this problem, and make the configuration as reliable as possible, is properly tuning the cluster, establishing a proper cluster size, controlling members status, preparing a backup and so on. ETCD has been pre-tuned on cloud providers with highly variable networks, and handles leader elections during network partitions, tolerating machine failure, including the leader.

ETCD is focused on four main characteristics:

- Being simple: this means, well-defined and direct. This requirement is achieved by using read and write values with curl and other HTTP libraries.
- Being secure: it provides an automatic TLS (Transport Layer Security) with optional client certificate authentication to provide the user the best security tool.
- Being fast: benchmarked 10,000 writes/sec.
- Being reliable: main concern of this kind of infrastructures, so it is properly distributed using the consensus algorithm before explained, Raft, to manage a highly-available replicated log.

Besides the configuration management, the following topic to take into account would be the communication with the ETCD, handled via the Raft consensus algorithm. The applications are able to read and write into this database, as well as track values written by other pair. The benefits that the user can obtain

from this kind of daemon are many, as providing a secure database that can resist the nodes failures in order to continue offering a service.

This communication is performed through an exposed REST-based API, which internally uses JSON on top of HTTP. The API may be used in two ways:

- **Direct**: through commands like cURL or wget.
- **Indirect**: through the command etcdctl, which is a specialized command-line utility also supplied by CoreOS.

There is a large list of libraries for the different programming languages that the user may choose. For example, it can be found the ETCD API for Java, Python, Ruby, C, C++, Node,... However, in this document it will shown only the provided tool, etcdctl, and cURL, from HTTP. In the Annex C there are listed the references for programming in these languages for implementing ETCD.

An important remark would be the type of data that ETCD stores from the nodes. Three types of resources can be found in this system:

1. **Key-Value resources**, the main objective of ETCD, as represents the key-value pairs requested and written by the clients in the key-value store.

2. **Setting resources**, which includes every management keys and directories regarding the security of the cluster, the authentication setting and the dynamic ETCD configuration, as leader elections or heartbeat.

3. **Permission resources**, that refers to users and roles in the user store, for the authentication procedure.

In the following sections, we will go though these insights to clarify the ETCD operation, giving special importance to its configuration and key-value management.

3.2 Objectives of ETCD

Besides its objectives of providing a simple, secure, and fast service, it is necessary to mention again that its principle main is to store metadata in a consistent and fault-tolerant way. Distributed systems use ETCD as a consistent key-value store for configuration management, service discovery, and coordinating distributed work. Common distributed patterns using ETCD include leader election, distributed locks, and monitoring machine liveness.

Basically, ETCD is designed as a general substrate for large scale distributed systems that will never tolerate split-brain operation and are willing to sacrifice availability to achieve this end. For this kind of systems, a cluster using ETCD is meant to provide consistent key-value storage with best of class stability, reliability, scalability and performance.

ETCD replicates all data within a single consistent replication group. For storing up to a few GB of data with consistent ordering, this is the most efficient approach. Each modification of cluster state, which may change multiple keys, is assigned a global unique ID, called a revision in ETCD, from a monotonically increasing counter for reasoning over ordering. Since there's only a single replication group, the modification request only needs to go through the Raft protocol to commit. By limiting consensus to one replication group, ETCD gets distributed consistency with a simple protocol while achieving low latency and high throughput.

Depending on the requirements of the application, it is better to use a database like ETCD or a NewSQL database:

- **NewSQL**: if the aim is to store data on the order of terabytes and above and the user would execute complex SQL-queries with a rich semantic, NewSQL database is a better solution than ETCD. On the other hand, this DB implies a long latency due to its scaling design and a bigger complexity for processing queries.
- **ETCD**: designed for providing a simple service, it is ideal for storing configuration data and distributed applications coordination, due to its little latency and easy of use. For storing more that gigabytes, it is recommended to implement a more scalable database, as NewSQL.

This is the reason why ETCD has been chosen for this project: the cluster size would not overpass the 7 members and the information shared will imply only cluster coordination, configuration data sharing and recovery from a majority failure.

3.3 Cluster Configuration

Different types of cluster configuration can be carried out. If the system has not started its operation, the parameters are established via flags at the time of the **bootstrapping**, this means, the startup of the cluster with few resources. These flags to configure the cluster are detailed in Annex A. At this moment, the administrator sets the members of the cluster and in which token they will be included. To perform the bootstrapping, the user can choose one of the following alternatives:

- **Static**, where the cluster is stopped; the administrator would perform offline operations. The cluster members are known, so their IP address will be used to start it up. The flag prefix for this mode is *-initial*.
- **Discovery service**, where the cluster is still not up, but the administrator here doesn't know the members IP address. The flag prefix for this mode is *-discovery*.

For **runtime reconfiguration**, it is performed a dynamic operation. As the cluster has already been bootstrapped, IP addresses are known, so neither *-initial* nor *-discovery* flags would be included here.

An important aspect to consider at the design the cluster is its size, because it will decide the fault tolerance of the peers. This configuration is one of those that can be done at **incremental runtime reconfiguration**, which allows users to update the membership of the cluster when it is already operating. However, it is important to take into account several aspects, like reconfiguration requests can only be processed when a majority of cluster members are functioning (it is highly recommended to always have a cluster size greater than two in production) and that it is unsafe to remove a member from a two member cluster (the majority of a two member cluster is also two). If there is a failure during the removal process, the cluster might not able to make progress and need to restart from majority failure, which makes this aspect a critical issue of ETCD. In this case, member flags (detailed in Annex B) are taken for performing changes in the cluster membership.

To sum up, there are many reason for a cluster reconfiguration, as follows:

- *Cycle or upgrade multiple machines*: sometimes it is necessary to perform maintenances tasks, so the best practice is to move the members at the same time.
- *Change the cluster size*: depending of the objective of the cluster, it is interesting to adapt the cluster size taking into account the trade-off; if the size is increased, better failure tolerance and better read performance, but worse write performance due to takes more time to replicate the new input data.
- *Replace a failed machine*: several reasons can produce a failure, like hardware problems, a data directory corruption and wrong configurations, among others. In those cases, it is fundamental to replace the failed node because it will affect to the general failure tolerance.
- *Restart cluster from majority failure*: as mentioned before, this is one of the highlights of ETCD functionalities. If the majority of the cluster is lost or all of the nodes have changed IP addresses, then manual action is necessary to recover safely. The basic steps in the recovery process include creating

a new cluster using the old data, previously shared, forcing a single member to act as the leader, and finally using runtime configuration to add new members to this new cluster one at a time.

3.4 Cluster Bootstrapping

As said in Section 3.3, the startup of a cluster could be perform in two manners: static or with discovery service. Using one or another depends on the administrator knowledge. In order to bootstrap a cluster statically, it is necessary to know exactly which will be their members, this is, their member IP addresses. But sometimes, this information is not known previously, so that the cluster must be started up thanks to the discovery service.

We will use for the explanation the three nodes indicated in Table 3.1. They are supposed to have installed the version 2.2.5 of ETCD, packeted for Linux.

Name	IP Address
Node1	10.0.3.2
Node2	10.0.3.3
Node3	10.0.3.4

Table 3.1: Example of a 3-member cluster

3.4.1 Static Bootstrapping

Due to the system administrator knows the cluster members, he can provide several flags indicating for each member its name and its URLs for interacting with peers and clients. These URL must include a valid IP address and port. ETCD has two official ports: 2380, for communication with peers, and 2379, for clients.

In addition, it has to be added that the initial state of the cluster is new, as it is being bootstrapped for the first time, and those URLs sent in the `-initial-cluster` flag must match with the initial advertise peer URL of each node, which can contain either domain names or IP addresses. This initial flag will be ignored once it has been execute, as it only works for bootstrapping a cluster the first time. In case of desiring to change one of these parameters, it has to be done via runtime reconfiguration, that will be explained in the following section.

Regarding the ETCD communication with clients, two flags are used: `-listen-client-urls` and `-advertise-client-urls`. As their names indicate, the first one works for listening client traffic and accept requests, while the last one advertises its URL to make the node reachable. Moreover, there are two URLs configured for listening: the IP address of the node itself and the localhost address.

Taking into account the nodes described in Table 3.1, the commands executed for starting up the cluster would be:

- In node1:

```
$       etcd -name node1 -initial-advertise-peer-urls http://10.0.3.2:2380 \
        -listen-peer-urls http://10.0.3.2:2380 \
        -listen-client-urls http://10.0.3.2:2379,http://127.0.0.1:2379 \
        -advertise-client-urls http://10.0.3.2:2379 \
        -initial-cluster-token etcd-example \
        -initial-cluster node1=http://10.0.3.2:2380,node2=http://10.0.3.3:2380,
                node3=http://10.0.3.4:2380 \
        -initial-cluster-state new
```

- In node2:

```
$       etcd -name node2 -initial-advertise-peer-urls http://10.0.3.3:2380 \
        -listen-peer-urls http://10.0.3.3:2380 \
        -listen-client-urls http://10.0.3.3:2379,http://127.0.0.1:2379 \
        -advertise-client-urls http://10.0.3.3:2379 \
        -initial-cluster-token etcd-example \
        -initial-cluster node1=http://10.0.3.2:2380,node2=http://10.0.3.3:2380,
               node3=http://10.0.3.4:2380 \
        -initial-cluster-state new
```

- In node3:

```
$       etcd -name node3 -initial-advertise-peer-urls http://10.0.3.4:2380 \
        -listen-peer-urls http://10.0.3.4:2380 \
        -listen-client-urls http://10.0.3.4:2379,http://127.0.0.1:2379 \
        -advertise-client-urls http://10.0.3.4:2379 \
        -initial-cluster-token etcd-example \
        -initial-cluster node1=http://10.0.3.2:2380,node2=http://10.0.3.3:2380,
               node3=http://10.0.3.4:2380 \
        -initial-cluster-state new
```

In case the administrator adds more than one URL in the parameter `-initial-advertise-peer-urls`, then that URLs have to be added to the flag `-initial-cluster` too. Remember that `-initial-cluster` flag must match with the `initial-advertise-peer-urls` flag of each node. In other case, ETCD would respond to this bootstrapping attend pointing that the URLs in both flags are different.

3.4.2 Public Discovery Service Bootstrapping

In the scenario where members are not known (for example, when the IP is not fixed, as in networks using DHCP or cloud providers) it is not possible to perform a static bootstrapping. This is why an alternative service was designed, for those cases IP addresses are unknown. It's called **discovery service**, to help connecting etcd instances together by storing a list of peer addresses, metadata and the initial size of the cluster under a unique address, known as **discovery URL**. For each discovery service, a new URL must be created (this service is only recommended for the bootstrapping moment, and after that phase a runtime reconfiguration would be performed instead).

For the creation of this URL, a existing cluster created previously by the user could be used as a reference. However, as our environment represents the development of a ETCD cluster from scratch, we have not a cluster reference.

For this case, CoreOs recommends to use its free service called **public discovery service** hosted at discovery.etcd.io. This service provides a discovery URL by performing a GET operation, obtaining a public ETCD cluster reference which would be used to help to build our own.

The first step is to ask for a cluster token by visiting *https://discovery.etcd.io/new?size=n*, where *n* represent the cluster size (by default, it is set to 3). The service will redirect *https://discovery.etcd.io/new?size=n* to ETCD cluster behind for the key at /v2/keys/_etcd/registry. It masks register key prefix for short and readable discovery URL.

Therefore, the administrator has to request this ETCD token by executing the following command from one of the servers, that would act as the cluster leader:

```
$ curl https://discovery.etcd.io/new?size=3
```

After executing the previous command, as output, we will get a new URL including the token. For example:

```
https://discovery.etcd.io/5043ed0b7df06bd5453a0487da2ada30
```

Then, as in the static operation, we have to execute the clustering flags on each member for bootstrapping the cluster. Notice that instead of the `-initial-cluster-token`, `-initial-cluster` and `-initial-cluster-state` flags, it is included only the *-discovery* one, with the URL provided in the previous response:

- In node1:

```
$       etcd -name node1 -initial-advertise-peer-urls http://10.0.3.2:2380 \
-listen-peer-urls http://10.0.3.2:2380 \
-listen-client-urls http://10.0.3.2:2379,http://127.0.0.1:2379 \
-advertise-client-urls http://10.0.3.2:2379 \
-discovery https://discovery.etcd.io/5043ed0b7df06bd5453a0487da2ada30
```

- In node2:

```
$        etcd -name node2 -initial-advertise-peer-urls http://10.0.3.3:2380 \
-listen-peer-urls http://10.0.3.3:2380 \
-listen-client-urls http://10.0.3.3:2379,http://127.0.0.1:2379 \
-advertise-client-urls http://10.0.3.3:2379 \
-discovery https://discovery.etcd.io/5043ed0b7df06bd5453a0487da2ada30
```

- In node3:

```
$        etcd -name node3 -initial-advertise-peer-urls http://10.0.3.4:2380 \
-listen-peer-urls http://10.0.3.4:2380 \
-listen-client-urls http://10.0.3.4:2379,http://127.0.0.1:2379 \
-advertise-client-urls http://10.0.3.4:2379 \
-discovery https://discovery.etcd.io/5043ed0b7df06bd5453a0487da2ada30
```

This permits to each member registering itself in the cluster. When everyone is included inside (remember that the size is specified at the beginning), ETCD starts to run in the new cluster.

In spite of resulting an interesting service and helpful when IP addresses are not known, it is not applicable to our case, using LXC. Public service discovery requires our instances in the subnet have **public IP addresses** or **Elastic IP addresses**. As we are working only with private IPs and have not configured any bridge to obtain a public IP address, public discovery service is not an option.

3.5 Cluster Administration

Once a cluster is bootstrapped, some maintenance tasks and upgrades could be needed if the cluster experiences changes, like failed nodes, erroneous configurations or a new member ready to be added. If any configuration is set in a wrong way, can lead to bigger issues, due to it is more difficult to fix it once the operation starts. Many parameters would need to be changed, and in some cases this is not a trivial task. This is why an administrator has to know the manner to properly reconfigure and manage the existing cluster.

In this section, the runtime reconfiguration design will be introduced, explaining the CoreOs recommendations to avoid frequent problems that ETCD could experiment. Moreover, some use cases of this reconfiguration will be shown in more detail, as well as other important aspects of this system.

3.5.1 Runtime Reconfiguration

ETCD provides several tools to perform a runtime reconfiguration of a cluster. For example, it is possible to upgrade multiples machines at the same time when they are running, or to replace a failed machine. However, CoreOs team recommends to be careful at the time of runtime reconfiguration, due to it is the moment that more errors are committed in a distributed system. If the administrator wants to remove a fallen member but commits a mistake, this error could cause a majority failure in the whole cluster, then the only solution would be to restart it.

To avoid this kind of issues, it is highly recommendable to perform the runtime reconfiguration in two phases. First of all, it is compulsory to inform the cluster that a new member is going to be added. Until the cluster answers with the consent of its members, this action cannot be performed. The next step is when the member adds itself to the cluster, using properly therefore explained flags `-initial-cluster` and `-initial-cluster-state` (see Annex A), keeping in mind that the value of the last one must be set to "existing".

When the member starts, it will check the cluster configuration matched with the previously indicated. If every parameter was correctly set, the new member will join the cluster with success.

There are several reasons why this design should be done in two steps:

- It forces the user to be more specific regarding the member addition, which makes more simple to understand the process and don't commit mistakes so easily. In addition, this it protects the cluster against unexpected cluster membership changes.
- As it is a two-phases process, if there is an error trying to modify the configuration, like providing an invalid ID, the task will fail at the first step, without affecting the cluster.
- If a new member wants to be added by error without informing the cluster, this one would not be accepted.

Besides this recommendations, because of reasons of safety and cluster management it is advisable to not perform frequently a runtime reconfiguration.

Another situation where a runtime reconfiguration is necessary is in the case of a nodes majority failure. This requires a new cluster, which would be started by means of recovering its state and information from an old data directory, that would act as a backup. In spite of the fact that it is possible to remove at the same time several failed member from the existing cluster, this method is consider by CoreOs as unsafe, because it splits the first phase of nodes consensus. The proper way to face a disaster recovery is explained in the section 3.5.5.

The last recommendation for the runtime reconfiguration refers not to use public discovery for this kind of operation, but for bootstrapping a cluster, as said in section 3.4.2. This is due to the aim of this service: start a cluster when the IP addresses are unknown. Once a cluster is bootstrapped, discovery service is not necessary anymore. Nevertheless, an administrator could use it to discover new IPs although the cluster is already up. CoreOs advises to avoid the discovery service if a bootstrapping is not being performed, because it would introduce external dependencies for the entire life-cycle of the cluster that could give problems if something in the network fails. In addition, public discovery service has to provide security mechanism against improper actions, which is a complicate task. In terms of workload, if the service is use for runtime reconfiguration, it will have to maintain a big amount of data that cannot be handled. So, if it is needed a discovery service to operate a runtime reconfiguration, it is preferable to build a private one to avoid this problems related.

Now the recommended design has been explained, we will go through several ETCD aspects about the cluster management.

3.5.2 Cluster Monitoring

One the cluster is running, one administrator task is to monitor its ETCD production. This is due to the certainty about the information the cluster is handling is correct. This measure is called health monitoring, because shows if the members of cluster are working properly (this means, they are "healthy") or not, and its testing can be done via HTTP at lowest level, using the JSON format:

```
$ curl -L http://127.0.0.1:2379/health
```

Another way to show the ETCD health is using the etcdctl command:

```
$ etcdctl cluster-health
```

The administrator can view the cluster's health with both commands, even though they return different outputs. Via HTTP, the result is the general health of the cluster, with an output like `"health": "true"` if it is healthy. Otherwise, with the last command the administrator can exactly know the health of each member, which at the end gives also the general state:

```
member 8211f1d0f64f3269 is healthy: got healthy result from
 http://127.0.0.1:1237
member 91bc3c398fb3c146 is healthy: got healthy result from
 http://127.0.0.1:22379
member fd422379fda50e48 is healthy: got healthy result from
 http://127.0.0.1:32379
cluster is healthy
```

Cluster monitoring can be also perform measures its runtime metrics, as latency, bandwidth and uptime. ETCD offers three types of statistics: about the **leader**, about the **store** operation and about the **node** itself. If the user take a look of the leader statistics, it could know the latency to each member of the cluster, as well as the number of failed and successful Raft RPC requests. These metrics are obtained by executing the following command:

```
$ curl http://127.0.0.1:2379/v2/stats/leader
```

Moreover, if this peer needs to know the information about the operations that have been performed during the cluster life, like creations, removals, settings and updates, it has to make the request it with:

```
$ curl http://127.0.0.1:2379/v2/stats/store
```

This command will also show the number or "*get*" and "*watch*" operations, but only those ones related to the local node. The last statistics to measure are about the node itself, where the user can know the identifier

of this node, its name, state (leader or follower), when the node was started, etc. In order to obtain these metrics, the peer has to execute:

```
$ curl http://127.0.0.1:2379/v2/stats/self
```

However, this kind of request can be only done using another node of the cluster, as it refers to the localhost address. If an external server wants to receive this information, it would have to ask directly to a

cluster node. Knowing the IP address of the interested node, it would be as follows:

```
$ curl http://10.0.3.2:2379/v2/stats/self
```

Another form that ETCD uses to monitor the metrics is the *Prometheus system*, where parameters like the proposals duration, the number of pending proposals waiting for commit or those ones which have failed, the total number of expired keys, current count of active watchers on a ETCD member, network issues, etc. are tracked. However, CoreOS warns that this metrics feature is considered experimental, so changes could be made in the future as it is still in development.

3.5.3 Data Directory

The first time that ETCD is started in a cluster, its configuration is stored in a data directory. These settings include the local member ID, cluster ID, and initial cluster configuration. The administrator would need to backup it for the case of a general failure, because it would help to restore the ETCD cluster. It us divided in two folders:

- **Wal**: the initial configuration settings are stored here. It is recommended to dedicate a disk for this sub-directory, as it would improve the throughput and cluster stability. This action is performed by setting a `-wal-dir` flag to point to a specific directory instead of the current one.
- **Snap**:the raft node's states are stores here.

About this directory, CoreOs wars that if a member's data directory is ever lost or corrupted, the user should remove this ETCD member from the cluster using etcdctl tool. Then, it can be added again with an empty data directory. Moreover, it is not recommendable to restart an ETCD member if the data directory backup is out-of-date, because that could cause inconsistences.

3.5.4 Cluster Debugging

In spite of the fact that it is a complicate task to debug a distributed system, CoreOs tries to provide tools to perform it in an easy manner. Therefore, in a scenario where the ETCD cannot be stopped, the developer would need to debug at runtime.

This action can be perform via the following command:

```
$ curl -XPUT http://127.0.0.1:2379/config/local/log -d '{"Level": LEVEL}'
```

Where "LEVEL" would take two values:

- **"DEBUG"**, which enables debug logging.
- **"INFO**: which is the default value, disables debug logging.

When this kind of debugging is enabled, developers have several debug variables to use via HTTP at /debug/vars in JSON format. These ones are used to debug an unexpected behavior, limiting the maximum number of file descriptors that ETCD can use, showing command lines arguments passed into ETCD, recording statistics about the memory allocator and solving debug low level Raft issues.

3.5.5 Members Management

Several times along this section we have mentioned that the cluster size is an important issue at the design of the network. On the one hand, it involves a trade-off between the failure tolerance and write tolerance, because as the cluster grows, more difficult it is to have a general failure, but more time for replicating the logs to the majority of members and for committing. On the other hand, the member's number of the cluster is recommended to be odd, because adding an extra member doesn't change the number needed for majority (with a cluster of 4 members, the majority is 3, but also for a 5-members cluster), but provides a better failure tolerance (with a 4-members cluster, only one can fail to maintain the majority; with 5 members, 2 members could do down without causing a problem). In this sense, it is advisable to have an ETCD cluster of 3, 5 or 7 members.

When a cluster is fully set and running and the administrator wants to change its size by adding or removing member, it would be done via runtime reconfiguration, as it has been explained at the beginning of this section. There are four possible operations: **adding**, **removing**, **updating** and **migration**. However, before making any change, the simple majority of ETCD members have to be available. Notice that it is not possible to perform several operations simultaneously: changes are done one at the time.

Adding a new member

This basic operation has two steps: using the etcdctl command (specifying the new member's name and the advertised peer URLs) or the members API (curl command with XPOST property), the new member is added to the cluster, and then the administrator has to start it with the new cluster configuration, this means, a list with the new and current members.

For example, imagine a cluster of 4 members, where the administrator wants to add a new one in order to improve the failure tolerance maintaining the same majority. The first step, where the cluster is informed about the new member, would be the following one:

```
$ etcdctl member add newmember http://10.33.1.10:2380
```

Then, the administrator must start it providing the specific environment variables for this new peer and the complete cluster. The action is executed in the node prepared to be added:

```
$ export ETCD_NAME="newmember"
$ export ETCD_INITIAL_CLUSTER="member1=http://10.33.1.11:2380,
        member2=http://10.33.1.12:2380,member3=http://10.33.1.13:2380,
        newmember=http://10.33.1.10:2380"
$ export ETCD_INITIAL_CLUSTER_STATE=existing
$ etcd -listen-client-urls http://10.0.1.13:2379
           -advertise-client-urls http://10.33.1.10:2379
           -listen-peer-urls http://10.33.1.10:2380
           -initial-advertise-peer-urls http://10.33.1.10:2380
           -data-dir %data_dir%
```

After both steps, the new member will finally join the cluster and start the communications with its peers. It us recommended to add members one by one, being careful in the process and checking that each one has been added before working with a new node.

Removing a member

A cluster administrator could be also interested in removing a member due to a failure, to maintain the cluster health. The needed parameter for this case is the member ID, indicated by the command "*etcdctl*

member list". From our previous example, if "newmember" has the ID *5e32elc3ae5f1e34*, to delete it the following command should be run:

```
$ etcdctl member remove 5e32elc3ae5f1e34
```

Then the cluster will announce that the member has been permanently removed. In the situation where the administrator wish to remove the leader of the cluster, it would be done taking into account that the cluster is going to remain inactive at the new leader election period.

Updating a member

Another case during the cluster management could be an update of member peerURLs, that means, the member IP address. Again, it is necessary to know the ID member to perform this operation and the new IP address. So, the update would be done as follows:

```
$ etcdctl member update 5e32elc3ae5f1e34 http://10.33.1.16:2380
```

Members migration

One of the main highlights of the runtime reconfiguration is the possibility of performing a migration of the ETCD cluster member to another machine without losing information and changing the member ID. The process consists in four steps with each member, one by one: stop its process, copy its data directory to the new machine, update the IP address to reflect the new machine and start ETCD on this new machine, with same configuration as before, because the data directory has been copied.

The following example shows these steps and its related commands, if we continue with the "newmember" peer:

1. Stop the member process:

```
$ ssh 10.33.1.16
$ kill 'pgrep etcd'
```

2. Copy the data directory to the new machine (10.33.1.17 would be the new IP address):

```
$ tar -cvzf newmember.etcd.tar.gz %data_dir%
$ scp newmember.etcd.tar.gz 10.33.1.17:~/
```

3. Update the IP address of the member to reflect the new machine:

```
$ etcdctl member update 5e32elc3ae5f1e34 http://10.33.1.17:2380
```

4. Start ETCD on the new machine, with same configuration:

```
$ ssh 10.33.1.17
$ tar -xzvf newmember.etcd.tar.gz -C %data_dir%
$ etcd -name newmember \
-listen-peer-urls http://10.33.1.17:2380 \
-listen-client-urls http:// 10.33.1.17:2379,http://127.0.0.1:2379 \
-advertise-client-urls http://10.33.1.17:2379,http://127.0.0.1:2379
```

Disaster Recovery

It has been largely talked how to avoid problems with the cluster enhancing the fault tolerance, that makes the ETCD cluster able to automatically resist to temporary failures, like machines rebooting, for example. In addition, a cluster of N members is design to tolerate up to (N-1)/2 permanent failures, this means, when a member cannot join the cluster anymore, due to a hardware failure or a disk corruption). If this tolerance limit is exceed and the cluster lose enough members, the ETCD administrator needs to perform a disaster recovery, which consists in three steps: **backup the datastore**, then **restore it** and finally **regenerate the cluster without data loss**.

Therefore, the administrator has to execute the etcdctl backup command in order to store the data directory on a running ETCD node, but must be taken into account that if users have been working in Windows environment, ETCD must be stopped before this execution. This instruction will overwrite the metadata contained in the backup to prevent the new node from inadvertently being added into an existing cluster. It would be as follows:

```
$ etcdctl backup \
 --data-dir %data_dir% \
 --backup-dir %backup_data_dir%
```

Once the backup has been saved, it will be restored by initializing a new member for a new cluster, pointing to this stored directory. This also sets the peer URL to the default. To start the ETCD again, it should be executed the following command:

```
$ etcd \
-data-dir=%backup_data_dir% \
-force-new-cluster \
...
```

The next step would be move the data back to the previous location, and it is necessary to shut down again the ETCD:

```
$ pkill etcd
$ rm -fr %data_dir%
$ mv %backup_data_dir% %data_dir%
$ etcd \
-data-dir=%data_dir% \
...
```

The last step once the cluster has been successfully started, it to restore the advertised peer URLs, as the current ones have been set to the default. This would be done as explained before, updating the cluster peers by the command `etcdctl member update`. Moreover, the administrator may want to add more members to the cluster, to restore resiliency as well.

3.6 Communication with ETCD

Following the previous sections, the ETCD cluster would be completely ready and set to start collecting information and sharing it with its peers and clients. As said at the beginning, this distributed database is based on a key-value store, where logs as connection details, cache settings, feature flags and other parameters are saved in a reliable way. In this section several manners to access to this data will be explained, as well as how to modify or track it.

3.6.1 Keys

ETCD provides two ways to read and write into its database: through the commands **etcdctl** and **cURL**, taking into account that the last one needs the *-L* flag. If a member wishes to add a key-value pair in the datastore, it would be done as follows:

- *etcdctl* command:

```
$ etcdctl set /firstkey value1
```

This instruction will create a key named "firstkey" with an associated value called "value1". When it is set, the ETCD will respond reapeating the value introduced. Let's see how to perform the same action with another command.

- *cURL* command:

```
$ curl -L -X PUT http://127.0.0.1:4001/v2/keys/firstkey -d value="value1"
```

Unlike the previous command, this one gets a more detailed answer from ETCD, with many attributes:

```
{"action":"set","node":{"key":"/firstkey",
"value":"value1","modifiedIndex":2,"createdIndex":2}}
```

The "action" attribute corresponds to the action request via PUT HTTP, in this case "set" because the aim is to introduce a new key. The object "node" has several attributes, as follows:

- *node.key*: represents the HTTP path, in this case "/first". As ETCD has an structure similar to a file-system, with directories, each key starts with a "/".
- *node.value*: represents the value of the previous key requested. In this case, it would be "value1".
- *node.modifiedIndex*: this is an ETCD index, an unique integer, incremented each time the ETCD processes a change, like "set", "delete", "update" "create", "compareAndSwap" and "compareAndDelete". The actions "get" and "watch" don't affect this index, as they don't perform any modification to the datastore.
- *node.createdIndex*: as the previous index, it is incremented by the changes made to the ETCD datastore, and represents the moment when the key was created, specifically the ETCD state member.

In this example, the key will not expire as not requirement has been set specifically. However, it would be interesting to establish a TTL to delete the key after a specified number of seconds. This action, expiring a key, can be performed only by the cluster leader, therefore if a member fails, its keys will remain until the re-connection. Using both commands, if we set another key that will expire in 10 seconds, it would be as follows:

```
$ etcdctl set /secondkey "value2" --ttl 10
$ curl -L -X PUT http://127.0.0.1:4001/v2/keys/secondkey?ttl=10 -d value=value2
```

With this new property, a two new fields are added to the "node" object: "expiration" and "ttl". The first one refers to the specific date at which the key will expire; and the other one, the expiring time in seconds (in this case, 10 seconds)

After these 10 seconds, users will not be able anymore to get the key, obtaining an error from ETCD like the following:

```
Error: 100: Key not found (/secondvalue) [32]
```

In addition, if this TTL is not needed anymore and the user wants to remove it to avoid the key expiration, an update operation should be executed:

```
$ curl -XPUT http://127.0.0.1:2379/v2/keys/secondkey -d value=bar d ttl= -d prevExist=true
```

In order to change the value of a key, it is as simple as performing again a PUT request with the new value, which will overwrite the previous one. The "createdIndex" and "modifiedIndex" are increased plus one due to the change, and a new attribute called "prevNode" appears. It represents the previous state of the peer in the same format as "node", before the change that have been made.

Working on a directory, a member could want to set several keys in order, like a queue, using the curl command with the action "POST". Let's see an example:

```
$ curl -XPOST http://127.0.0.1:2379/v2/keys/queue -d value=FirstValue
$ curl -XPOST http://127.0.0.1:2379/v2/keys/queue -d value=SecondValue
```

If the user needs to know the key queue from the beginning until the end, it has to specified that wants a sorted list. ETCD will return the previous values in order:

```
$ curl -s 'http://127.0.0.1:2379/v2/keys/queue?recursive=true&sorted=true'
```

On the other hand, if it simply needs the list, but from the actual value to the first one set, it has to execute the order shown before, without any property:

```
$ curl -s 'http://127.0.0.1:2379/v2/keys/queue
```

In some cases, the user could be interested in setting a key from a file because it contains configuration parameters (a txt, JSON or XML document, for example). It can be performed directly by executing the following command:

```
$ curl -XPUT http://127.0.0.1:2379/v2/keys/configfile --data-urlencode value@configfile.txt
```

To finish the topic about keys creation, there is a last service in ETCD called "Compare-and-Swap" (CAS), which consists in setting a key only if the client provides the same conditions as the current set ones. These conditions would be:

- *prevValue*, which compares with the previous key value.
- *prevIndex*, which compares with the previous key modifiedIndex.
- *prevExist*, which checks if the key exists. If it is *prevExist=true*, the key is updated if the condition is achieved; if it is *prevExist=false*, it is created.

For example, a case where the "prevExist" condition is used:

```
$ curl -XPUT http://127.0.0.1:2379/v2/keys/color?prevExist=false -d value=red
```

When a user perform this operation, it would set a new key called "color" only if it didn't exist at that moment. If it is already at the ETCD datastore, the order will be ignored. It is an interesting tool to check that the keys stored have been created, and for setting them in the case that they don't exist.

Once we have several keys stores in ETCD, the users will be able to access to them and watch, update and delete it. If there is a node interested in an specific key, it will make a GET request using the etcdctl command or cURL:

- *etcdctl* command:

```
$ etcdctl get /firstkey
```

The direct answer from ETCD will be "value2", without more details. Let's see what happens if the user make the request with the other command.

- *cURL* command:

```
$ curl -L http://127.0.0.1:2379/v2/keys/firstkey
```

In this case, as seen before, the cURL command returns a more complete answer. Specifically, that shown when the user set the key, but with a different action, "get":

```
{"action":"get","node":{"key":"/firstkey",
"value":"value1","modifiedIndex":3,"createdIndex":3}}
```

Every cluster member trying to access to this key have to receive the same value and attributes.

To write into ETCD, performing a PUT operation:

```
$ curl -X PUT http://127.0.0.1:2379/v2/keys/thirdkey -d value="value3"
```

After a time, the user could consider that its key has to be removed because it is not useful anymore. If a TTL period has not been set, this action can be executed by running again:

- *etcdctl* command:

```
$ etcdctl rm /firstkey
```

- *cURL* command:

```
$ curl -X DELETE http://127.0.0.1:2379/v2/keys/firstkey
```

Here it is also possible to perform a removal operation if a condition is achieved, as for creating keys. The conditions for a "Compare-and-delete" condition are:

- *prevValue*, which compares with the previous key value.

- *prevIndex*, which compares with the previous key modifiedIndex.

For example, if a user is interested in deleting the key "color" only if its value is "red", it would execute:

```
$ curl http://127.0.0.1:2379/v2/keys/color?prevValue=red -XDELETE
```

After this brief explanation about how the cluster members can manipulate the keys, an important tool of ETCD should be presented: how to track them. The clients of ETCD will be interested in a particular key so that they will try to watch any changes that it experiments. In this way, it can be said that many events can trigger a change, including a new, updated, deleted or expired key. The simplest manner to perform it is via a GET operation, using *cURL*, or through the *etcdctl* command.

- *etcdctl* command:

```
$ etcdctl watch --recursive /secondkey
```

This action, after waiting for a change, will return from ETCD the new value introduced by another cluster peer.

- *cURL* command:

```
$ curl http://127.0.0.1:2379/v2/keys/randomkey?wait=true
```

In this case, ETCD will answer with the operation details, as the new and old values, the incremented index, etc. Moreover, if the user is interested in the history of the key, this means, the number of changes that the key suffers, it is possible to execute a command that watches with specific key until a known index:

```
$ curl 'http://127.0.0.1:2379/v2/keys/randomkey?wait=true&waitIndex=40'
```

This member will watch the key "randomkey" until it had reached the modifiedIndex=40, when ETCD will return the new and previous value and stop watching. This tool could be useful to track the keys until a point, instead of only obtaining the following change in time. However, the peer could be interested in a specific past index because it was not watching the key at that moment. In that case, it has only to specify the number in the instruction in the same way as the last command shown, and ETCD will immediately answer with the value changed at that time. This operation cannot be perform always, as ETCD only keeps the responses of the most recent 1000 events across all stored keys. In the case of a user that have missed 1050 events, and tries to recover the key changed value at the modifiedIndex=5, an error will be returned saying: "The event in requested index is outdated and cleared", due to the requested history has been deleted. In order to start watching the key, it has to be indicated the X-ETCD-Index+1, as it will be greater than the modifiedIndex. By executing the following command, it is possible to know the current ETCD index of a key:

```
$ curl 'http://127.0.0.1:2379/v2/keys/randomkey' -vv
```

Sometimes, ETCD could give an empty answer to a watch operation that have been running, due to the server has closed the member polling request before observing an event. This can be due to an error caused by a timeout or because the server has been turned off or restarted. CoreOS recommends to take into consideration these kind of issues in order to retry the watch request again.

3.6.2 Directories

When the keys storage is big enough, it could be difficult to handle a cluster monitoring tracking several keys, and the user would prefer to control a directory of keys, placed there due to its similarity or relation, instead of individual keys. These directories can contain a set of keys but also other directories. In this section, we will see how to work with directories, similarly to keys management.

The first step that would be taken is the directory creation. This is not always necessary, as a directory can be set as easily as placing key inside it. At a PUT request, the user has to name de key with the format "/directory/key", and ETCD will recognize that the first term is a directory and the last one, a key placed inside. The command would follow the same format as for the key creation:

```
$ curl -XPUT http://127.0.0.1:2379/v2/keys/newdirectory/keyinside -d value=randomval
```

However, sometimes the user could be interested in creating an empty directory, without any key stored. In these cases, the operation to perform is similar as at the key topic, only indicating that the objective is a directory:

```
$ curl -XPUT http://127.0.0.1:2379/v2/keys/emptydirectory -d dir=true
```

ETCD will answer as before, indicating that it has been created a directory and without a value field, because it is a "folder":

```
{"action":"set","node":{"createdIndex":15, "dir": true,
"key":"/emptydirectory", "modifiedIndex":15}}
```

In the first case, creating a directory by placing a key inside, ETCD confirms the request in a manner that the user could find tricky, because it is the same format as at the key setting. It can be seen at the first example:

```
{"action":"set","node":{"createdIndex":14,
"key":"/newdirectory/keyinside", "modifiedIndex":14, "value":randomval}}
```

If the user doesn't notice that the format of the "key" is different, he could think that the directory has not been created. But, if the ETCD keys are listed, the directory is perfectly indicated, with its keys inside:

```
$ curl http://127.0.0.1:2379/v2/keys/?recursive=true
```

```
{
        "action": "get",
        "node": {
        "key": "/",
        "dir": true,
        "nodes": [
                {
                        "key": "/newdirectory",
                        "dir": true,
                        "nodes": [
                                {
                                "key": "/newdirectory/keyinside",
                                "value": "randomval",
                                "modifiedIndex": 14,
```

```
                                    "createdIndex": 14
                                    }
                                            ],
                                    "modifiedIndex": 14,
                                    "createdIndex": 14
                    }
            ]
        }
}
```

If the peer had used the usual command for displaying the usual commands, only the keys and directories of the root data directory would have shown, without indicating if there is a key stored in those ones.

A possibility when creating directories -or keys- is to make them hidden. This action would by perform by simply adding a prefix before its name:

```
$ curl -XPUT http://127.0.0.1:2379/v2/keys/_emptydirectory -d dir=true
```

Any node can access to this directory if it knows its name, but it is not possible by executing the listing commands previously shown (only the normal keys and directories would be indicated).

There are two manners for removing a directory, depending of its content (empty or not). For an empty directory, the procedure is as follows, using the DELETE action adding the "dir=true" parameter:

```
$ curl -XDELETE 'http://127.0.0.1:2379/v2/keys/randomdir?dir=true'
```

If it contains keys inside, the "recursive=true" has to be added, instead of "dir=true":

```
$ curl -XDELETE http://127.0.0.1:2379/v2/keys/randomdir?recursive=true
```

3.7 Authentication procedure

Another way to take advantage of all the tools and features of ETCD is to use the authentication system that it offers, based on users whom are assigned different permissions: **reading**, **writing**, or **both**. In this sense, there can be two types of users: **root**, which is mandatory to be activated due to terms of security, before enabling the authentication mode, and **normal**, that are the other members accessing to ETCD. This root user has linked the root role, which permits him to globally access to every directory and key with a read-write permission and to enable/disable the authentication mode. It was created because of the necessity of a user that could perform a recovery task in case of failure, as well as being able to manage the cluster configuration in terms of membership operations.

Regarding these roles, as mentioned, there are two by default: root and guest. While root is addressed to the management tasks, guest permits to perform those actions that don't require an authentication. Other roles can be created regarding the different directories and keys of the database. For example, the administrator could decide that the directory "ETCDInfo" should be modified by only a few users with "read-write" permissions, this means, full access, and other part of users (like managers without admin permissions) could read these keys but not adding new ones.

The first step to start the authentication feature could seem to enable it. However, leaving this as a last step and first of all setting the users and roles that will be used, may be the easier way to proceed, due

to if something is wrongly configured the administrator could go back to the initial setting without further problems. So, the first step would be the creation of several users that represent groups of clients with different grants. In order to start with this creation, the administrator executes:

```
$ etcdctl user add User1
```

A prompt would appear to introduce a password for this user, and later it could be changed as follows:

```
$ etcdctl user passwd User1
```

Again, another prompt would appear to change the initial password. If the authentication is enabled, the user must have enough permissions to perform this action, and that's why it is better to prepare the authentication system before activating it in ETCD.

Once the administrator has created every user that will work through the ETCD system, it is the moment for designing the roles that those users would have. For example, if they will work over the "ETCDInfo" directory, there would be three types of roles: for clients that will only write over this directory, for managers that only would have read access to it and for administrators with full access. Following this description, it will be performed as follows:

```
$etcdctl role add Client_role
$etcdctl role add Manager_role
$etcdctl role add Admin_role
```

Next, it is assigned the permission associated to each role:

```
$ etcdctl role grant Client_role -path '/ETCDInfo/*' -write
$ etcdctl role grant Manager_role -path '/ETCDInfo/*' -read
$ etcdctl role grant Admin_role -path '/ETCDInfo/*' -readwrite
```

The '*' and the end of the path means that that directory and the keys under it are granted to holders of this role. If there is not that symbol, the user only has permissions over that key. The administrator could wish to grant permission to every key starting by "ETCD" (for example, "ETCDInfo", "ETCDAdministration", "ETCDStorage", etc). To do so, it would execute without the last slash:

```
$ etcdctl role grant Admin_role -path '/ETCD*' -readwrite
```

After establishing the roles, the next step is to associate these ones with the proper users. It can be linked as many roles as desired at the same time:

```
$ etcdctl user grant User1 -roles Client_role1, Client_role2
$ etcdctl user grant Admin1 -roles Admin_role
```

Once everything is configured, the administrator can do some checking in order to be sure that it is correctly set. To see which permissions each role has or which roles has the user associated, the instructions are:

```
$ etcdctl role get Client_role1
$ etcdctl user get User1
```

A general vision can be prompt to have an overview of every role and user created:

```
$ etcdctl role list
$ etcdctl user list
```

After the configuration, several situations can occur:

1. Users don't used where the administrator has to delete their account:

   ```
   $ etcdctl user remove User12
   ```

2. Roles don't used where the administrator has to delete their definition:

   ```
   $ etcdctl role remove Client_role12
   ```

3. Roles with wrong permissions where the administrator has to revoke them:

   ```
   $ etcdctl role revoke Clientrol5 -path '/ETCDInfo/*' -readwrite
   ```

At this point, the system is completely set, and the last action of the administration part will be to enable the authentication in the system. This cannot be possible before creating a root user for security issues, as mentioned before, so it is proceed as follows:

```
$ etcdctl user add root
```

The administrator would introduce the root password here, and then:

```
$ etcdctl auth enable
```

At this moment, the authentication is working in ETCD. To reverse this action, a reciprocal command is provided:

```
$ etcdctl -u root:rootpassword auth disable
```

ETCD also recommends to check what are the permissions granted to the users with the guest role, this means, the authentication users:

```
$ etcdctl -u root:rootpassword role get guest
```

From the client part, to be logged into the system for obtaining a key, the following commands are used:

```
$ etcdctl -u user:password get ETCDInfo
```

In the case that the user doen't want to show its password, it can be introduced in a prompt:

```
$ etcdctl -u user get foo
```

This version of ETCD only supports "basic Access authentication", which is a method for a HTTP user agent to provide a user name and password when making a request. It is the simplest manner to enforce access control due to cookies, session indentifiers or login pages are not needed, only a HTTP Authorization header.

Chapter 4

ETCD Practice

4.1 Creation of the scenario

Naturally, the first step we will take in this practice is to create a scenario in which to perform tests to understand how the ETCD works and all its associated tools. It is important to remember all the recommendations the developers have proposed for the correct operation of the cluster. The design should take into account factors such as the optimum size of the cluster, depending on the type of application where it will be implemented and the number of clients that will have to support, and also the number of nodes that would be ideal to include. So, to begin, we should answer these questions:

- **Why is it recommended that the number of members of the cluster be odd? What would be more functional, a small cluster or a larger cluster? Why?**

Then, we will proceed to the creation of a nodes cluster and also the installation of every necessary tool in order to start using ETCD.

4.1.1 Implementing a nodes cluster

As mentioned, Linux Containers (LXC) will be used to provide us with a virtual environment with its own space of processes and networks. This will permit us to perform the tests with ETCD without affecting the physical machine. These LXC can be used in a privileged way, with the root user, as well as without privileges, executing the lxc command as a normal user. However, as unprivileged containers are more limited, we will use the root user. Using root carries the associated danger that an error when designing can trigger a problem that is difficult to solve. As for the operating system used, it will be Ubuntu, based on GNU / Linux, and whose desktop environment is called Unity. Thus, the command to download the package "lxc" is as follows:

```
sudo apt-get install lxc
```

Once downloaded, it's time to start implementing our cluster of nodes. To not add extra complexity, we will create up to 5 Linux containers that will work as servers in our system. To do this, it is necessary to log in as superuser with `sudo su` and execute the following instructions:

```
$ lxc-create -t ubuntu -n server1
$ lxc-create -t ubuntu -n server2
$ lxc-create -t ubuntu -n server3
$ lxc-create -t ubuntu -n server4
$ lxc-create -t ubuntu -n server5
$ lxc-create -t ubuntu -n client
```

In this way we have created a group of 5 nodes and one more extra, that we will use as an external client. However, we cannot access to any server now, nor they have any associated IP address. This is due to they have not been initialized. Every user can know its status by executing the following state:

```
$ lxc-info -n server1
```

However, this function only indicates whether the node is on or off. It is more interesting the following instruction, which indicates the status of all nodes, in addition to their IP address, if they belong to a group and if they have the AUTOSTART function enabled:

```
$ lxc-ls -f
```

As we can verify, all the nodes we just created are off, so we need to raise them to work with them. In this way, we execute:

```
$ lxc-start -n server1
$ lxc-start -n server2
$ lxc-start -n server3
$ lxc-start -n server4
$ lxc-start -n server5
$ lxc-start -n client
```

If we check again its status, we will see that "RUNNING" tag appears, and with an associated IP. For convenience, we recommend to write them down, since their IPs will be used later for the ETCD configuration. The situation of the cluster would be as detailed in Figure 4.1.

```
NAME    STATE   AUTOSTART GROUPS IPV4       IPV6
client  STOPPED 0         -      -          -
server1 RUNNING 0         -      10.0.3.194 -
server2 RUNNING 0         -      10.0.3.114 -
server3 RUNNING 0         -      10.0.3.215 -
server4 RUNNING 0         -      10.0.3.77  -
server5 RUNNING 0         -      10.0.3.23  -
```

Figure 4.1: Server IP addresses

In case something goes wrong during the configuration, as we are working with LXC, it is easy to remedy. Being a failure at startup, the simplest solution would be to delete the wrong server and re-create it from scratch (taking into account that the IP will change). The instruction to remove the server named server1 would be as follows:

```
$ lxc-destroy -n server1
```

As a confirmation, the console returns the message "Destroyed container server".

Now that we have the cluster created, it is time to install ETCD on virtual servers.

4.1.2 Installing ETCD

To proceed with the installation of the tool on each node of the cluster, it is necessary to access them. This operation is done by the command "lxc-attach" that allows to execute instructions inside the virtual server with the privileges of the user root. So, we introduce the following instruction in the window of commands and accede to the server1:

```
$ lxc-attach -n server1
```

Once inside, we will proceed to the installation of ETCD. This can be done either by downloading it from the repository found in GitHub (*coreos/etcd*) or by obtaining the package from the command window. For reasons of simplification, we will do this last. This way, we look for the package *etcd* and install it in the five servers:

```
$ sudo apt-get install etcd
```

If we type "etcdctl" in the commands window, it will be shown some interesting details about this command-line client. The default version installed would be v. 2.2.5, one version before that last one, which is currently in an experimental phase and being proved. A sort of commands associated to etcdctl are listed, which will be used during this practice.

Another package useful to develop ETCD would be "*cURL*", which is described as a command-line tool for getting or sending files using URL syntax. This means, for example, for downloading content from Internet. The supported Internet protocols supported are HTTP, HTTPS, FTP, FTPS, SCP, SFTP, TFTP, LDAP, DAP, DICT, TELNET, FILE, IMAP, POP3, SMTP and RTSP, among others. Again, to download this package we have to execute the order:

```
$ sudo apt-get install curl
```

- **Why do you think this cURL tool is useful for the ETCD case? How would be the information acceded?**

After obtaining these tools in every server (not client), we are ready to start the ETCD cluster configuration.

Important note

```
When the ETCD package is installed, it starts to work. This can be checked by looking to
the used ports:

$ netstat -tlnp

As we prefer to start our ETCD cluster with its proper initial configuration, it would be
 a good idea to stop the process. In order to do so, the executed order will be:
$ systemctl stop etcd.service
```

4.2 Cluster Configuration

4.2.1 Initial set up

Now we have a set of 5 nodes, but they are not interconnected sharing the same ETCD database. This step must be taken very carefully due to it is easy to write a wrong IP address or port. That's why the user needs to understand which port has to be used in each flag (see Annex I and II, clustering and member flags). In addition, it is mandatory to write the IP addresses properly in a consistent manner, this means, to match the IP address written in the flag "`-initial-advertise-peer-urls`" with the one added in "`-initial-cluster`". Another common mistake is setting "advertise-client-urls" to localhost or leave it as default when the users wants the remote clients to reach ETCD. As the cluster we are creating is new, the `-initial-cluster-state`" flag will be defined with the value "new" .

On each node, execute the following instruction (that have to be adapted) to initialize the ETCD cluster:

```
$ etcd  -name nodename -initial-advertise-peer-urls http://IP:port \
-listen-peer-urls http://IP:port  \
-listen-client-urls http://IP:port,http://IP:port \
-advertise-client-urls http://IP:port   \
-initial-cluster-token etcd-cluster \
-initial-cluster nodename=http://IP:port,nodename2=http://IP:port,nodename3=http://IP:port \
-initial-cluster-state new
```

If we adapt this order for server1, the result would be:

```
$ etcd  -name server1 -initial-advertise-peer-urls http://10.0.3.194:2380 \
-listen-peer-urls http://10.0.3.194:2380  \
-listen-client-urls http://10.0.3.194:2379,http://127.0.0.1:2379 \
-advertise-client-urls http://10.0.3.194:2379   \
-initial-cluster-token etcd-cluster-practica \
-initial-cluster \ server1=http://10.0.3.194:2380,server2=http://10.0.3.114:2380, \
server3=http://10.0.3.215:2380,server4=http://10.0.3.77:2380,server5=http://10.0.3.23:2380
\
-initial-cluster-state new
```

- **Which orders should be executed on server2, server3, server4 and server5?**
- **Which are the official ports of ETCD? What are they used for?**

The ports used by our LXC container can be obtained (Figure 4.2) by executing the order:

```
root@server1:/# netstat -tlnp
Conexiones activas de Internet (solo servidores)
Proto  Recib Enviad Dirección local        Dirección remota      Estado       PID/Program name
tcp        0      0 127.0.0.1:2379         0.0.0.0:*             ESCUCHAR    466/etcd
tcp        0      0 10.0.3.194:2379        0.0.0.0:*             ESCUCHAR    466/etcd
tcp        0      0 10.0.3.194:2380        0.0.0.0:*             ESCUCHAR    466/etcd
tcp        0      0 0.0.0.0:22             0.0.0.0:*             ESCUCHAR    186/sshd
tcp6       0      0 :::22                  :::*                  ESCUCHAR    186/sshd
root@server1:/#
```

Figure 4.2: Network information

```
$ netstat -tlnp
```

After running the proper instructions on each server, ETCD starts showing some data at the command line. Let's take a moment to analyze it:

- **Which version of ETCD has been installed?**
- **Can you identify the consensus algorithm that is being used?**
- **Which is the heartbeat period? And the election timeout?**
- **Can you identify which the current term is? Why were the servers at different terms?**
- **Which server is the leader of the cluster? Write its ID.**
- **Which is the *ETCD_INITIAL_CLUSTER-STATE* value?**

A good manner to check if the cluster configuration has been properly set and every node is running without errors, is to obtain the cluster health. Open an additional commands window to attach again the server1 and execute the following command:

```
$ etcdctl cluster-health
```

The output would be similar to Figure 4.3. Now we can check the IDs of each node, as the IP address is also returned, and to realize which the cluster leader is.

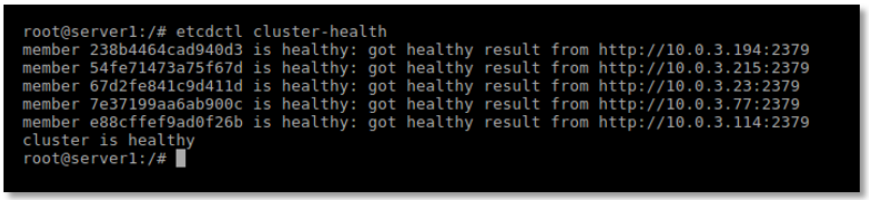


Figure 4.3: Cluster Health

If we are just interested in the current cluster health, but not in the details of each node, it can be used an order where, at lowest level, ETCD exposes health information via HTTP in JSON format:

```
$ curl -L http://127.0.0.1:2379/health
```

Since the cluster is supposed to be correctly configured and working, the output would be `"health":"true"`.

Another way to check the membership of the cluster is by using the following instruction, which will communicate to the user the ID, name, peerURLs and clientURLs of each node:

```
$ etcdctl member list
```

Now, stop the ETCD process of the leader and check what happens at the command windows of the other nodes.

- **Which is the term now? And the new leader?**

- **Which message is being prompted repeatedly? Check the cluster health. How could it be solved, if that node is fallen?**
- **Run ETCD again, which message is prompted now? Which is now the cluster health?**

4.2.2 Members management

Imagine again a similar situation as the described in the previous section, when a node stops working properly. At the command windows it is shown the message: "rafthttp: the connection to peer ? is unhealthy". In spite of still functioning, for safety reasons it is advisable to remove the fallen node of the cluster.

Stop the ETCD process of server2, taking into account that if it is the leader, the cluster will be inactive while a new leader is elected. Take the ID of the fallen node and run the following command to remove it from the cluster (replace the last field with the server2 ID):

```
$ etcdctl member remove e88cffef9ad0f26
```

If everything has gone correctly, the message "Removed member e88cffef9ad0f26 from cluster" will be prompted, and again the cluster is defined as healthy. Now we have a 4-nodes cluster, but as at the beginning, it is advisable to have an odd number of servers. So, we will add once again another node to gain in fault tolerance. Create a new server called "server6" and install the ETCD inside it. Take the IP of server6 (check Figure 4.4) and execute in a server of the cluster:

```
$ etcdctl member add server6 http://10.0.3.51:2380
```

```
root@sonia-ThinkPad-Yoga-460:/home/sonia# lxc-ls -f
NAME    STATE   AUTOSTART GROUPS IPV4       IPV6
client  STOPPED 0         -      -          -
server1 RUNNING 0         -      10.0.3.194 -
server2 STOPPED 0         -      -          -
server3 RUNNING 0         -      10.0.3.215 -
server4 RUNNING 0         -      10.0.3.77  -
server5 RUNNING 0         -      10.0.3.23  -
server6 RUNNING 0         -      10.0.3.51  -
root@sonia-ThinkPad-Yoga-460:/home/sonia#
```

Figure 4.4: New server added

Now the cluster of nodes has been informed about a new server that wants to be added. However, this order doesn't imply that server 6 is already working with the other nodes. To check this, look at the cluster health and the member list.

- **Is the clusther healthy? Why?**
- **Why if server6 is marked as a member of the cluster, it is still not functioning with ETCD? Which information does the cluster needs from this node?**
- **Which is the ETCD_INITIAL_CLUSTER-STATE value now?**

The next step to add a new server in the cluster, it is to start it with the new cluster configuration, including a list of the updated members. This information is provided by ETCD after executing the adding

order. In Figure 4.5 we can see that some environmental variables have been defined: ETCD_NAME, ETCD_INITIAL_CLUSTER and ETCD_INITIAL_CLUSTER_STATE.

```
root@server1:/# etcdctl member add server6 http://10.0.3.51:2380
Added member named server6 with ID f04d8a14261228f5 to cluster

ETCD_NAME="server6"
ETCD_INITIAL_CLUSTER="server1=http://10.0.3.194:2380,server3=http://10.0.3.215:2
380,server5=http://10.0.3.23:2380,server4=http://10.0.3.77:2380,server6=http://1
0.0.3.51:2380"
ETCD_INITIAL_CLUSTER_STATE="existing"
root@server1:/#
```

Figure 4.5: Server6 environmental variables

Using the information provided, we export these variables in server6 to initialize it in the ETCD cluster. Execute the following orders, adapting them correctly:

```
$ export ETCD_NAME="server6"
$ export ETCD_INITIAL_CLUSTER="server1=http://10.0.3.194:2380,server3=http://10.0.3.215:2380,
  server5=http://10.0.3.23:2380,server4=http://10.0.3.77:2380,server6=http://10.0.3.51:2380"
$ export ETCD_INITIAL_CLUSTER_STATE="existing"
```

Now, it is time to start the ETCD process in server6. Remember, first of all the ETCD process predefined by default at the installation must stopped (to do so, check the ETCD installation section). Then, run the order:

```
$ etcd  -listen-client-urls http://10.0.3.51:2379 \
                -advertise-client-urls http://10.0.3.51:2379  \
                -listen-peer-urls http://10.0.3.51:2380 \
                -initial-advertise-peer-urls http://10.0.3.51:2380 \
                 -data-dir %data_dir%
```

- **Which message is shown now at the ETCD command window about the new member?**
- **Which is the current term?**

Let's monitor now the statistics of our members. First, we will check the metrics of our new node: serve6. It can be done from other node of the cluster, as they are interconnected thanks to ETCD. So, we will go inside server1 and execute the following order indicating the server6 IP address:

```
$ curl http://10.0.3.51:2379/v2/stats/self
```

- **With this information, can you identify the leader of the cluster? Clue: obtain the ETCD member list.**
- **Which is server6 state?**

Next, we will take a look to leader metrics, as it has been identified by server6. In my case, it is server3. Adapt the IP address and run this order:

```
$ curl http://10.0.3.215:2379/v2/stats/leader
```

- **Which is the state of the server?**
- **Which differences can you appreciate in the metrics? Why are they different compared to server6?**

Now, change from the last order "self" to "leader" and execute the new instruction:

- **Of whom are the metrics described? Which is the only node without information?**
- **Check server6 metrics, had this node some problems in the communication with the leader? If so, what do you think is due to?**

In addition, if we want to know the metrics of the node we are working with, but we don't remember the IP address, we can just execute:

```
$ curl http://127.0.0.1:2379/v2/stats/self
```

4.2.3 Authentication guide

ETCD provides an interesting feature about authentication tools. As we are working with a database, it is something normal not to have access to the whole information stored inside. Users with more privileges can be defined by assigning different roles (those predefined by default or customize). As usual, there is a special user, called "root" with full permissions over the system. It is mandatory to create it due to, if there is a general failure, someone has to perform recovery tasks.

Prior to entering in definitions of users or roles, let's create some directories in ETCD in order to work with them. For example, we can set up three about public information, with free access, another about management information and the last one about ETCD itself configuration. After that, we will establish the permissions over these three directories in terms of writing or reading privileges. Execute the following

instructions to create the directories:

```
$ etcdctl mkdir Public_Information
$ etcdctl mkdir Management_Information
$ etcdctl mkdir ETCD_Information
```

If the user wants to check if they have been correctly created, he has just to run:

```
$ etcdctl ls
```

The first logical step at the authentication setting up would seem to be to enable this feature. However, it is recommended not to do so, due to the fact that if something is wrongly configured, it could cause a critical damage to ETCD database. Therefore, first of all we will create four types of users: clients, managers, special users and ETCD administrators.

Execute the following instructions:

```
$ etcdctl user add Client1
$ etcdctl user add Client2
$ etcdctl user add Manager1
$ etcdctl user add Admin1
$ etcdctl user add Special
```

For each one of these users, it will be shown a prompt asking for a password. To list every user created, run this order:

```
$ etcdctl user list
```

Once the users have been created, the next step is to create different roles to be assigned to them. As mentioned before, these roles are related to Write/Read privileges over a specific directory. So, there will be three customize roles:

```
$ etcdctl role add Client_role
$ etcdctl role add Manager_role
$ etcdctl role add Admin_role
```

Now it is time to decide which role can do what. For the "ETCD-Information" directory, only admin can read/write, and the managers can only read the ETCD configuration, but no editing it. In addition, we can grant the keys under the directory to holders of this role by adding a "*" at the end of the directory path. So, it would be:

```
$ etcdctl role grant Manager_role -path '/ETCD_information/*' -read
$ etcdctl role grant Admin_role -path '/ETCD_information/*' -readwrite
```

Configure the roles for:

- **Public-Information: Managers and Clients can both read and write, but Admins can only read the information.**
- **Management-Information: Managers can read and write and Admins can only read the information, but Clients cannot do anything.**

Again, the ETCD user can review the list of roles created by executing:

```
$ etcdctl role list
```

- **How many roles are listed? What does the last one stand for? Why is there, if we have not created that one?**

We can also look into each role to know which are the permissions associated. For example, take a look of Admin-role. The output must be as it in shown in Figure 4.6.

```
$ etcdctl role get Admin_role
```

Users and roles must be linked. In order to grant clients, managers and administrators with the different privileges we configured, execute:

```
$ etcdctl user grant Client1 -roles Client_role
$ etcdctl user grant Client2 -roles Client_role
$ etcdctl user grant Admin1 -roles Admin_role
$ etcdctl user grant Manager1 -roles Manager_role
$ etcdctl user grant Special -roles Admin_role,Manager_role
```

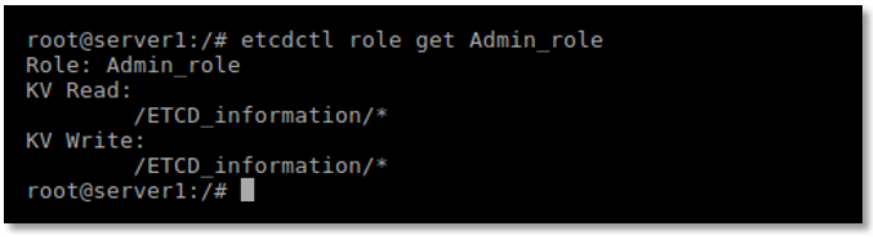


Figure 4.6: Admin role permissions

Once everything is configured, we will check if everything is correctly set. To see which roles has the user associated, the instruction is:

```
$ etcdctl user get Special
```

We can say that the authentication has been completely set, and the last step to be taken would be to enable this feature in the ETCD cluster. The instruction to enable authentication is:

```
$ etcdctl auth enable
```

- **Which message is prompt? Why do you think they request to create that user?**

Create the root user and execute again the previous instruction, enabling the authentication.

```
$ etcdctl user add root
```

Try to execute now one of the precious orders we have been using in this section. For example, try to get the user list of ETCD.

- **Which message is prompted?**

Now, only the root user has enough permissions to operate in the ETCD configuration. Due to fact, it is necessary to add a new parameter: the username. In order to get the user list, execute the following order and write the root password:

```
$ etcdctl -u root user list
```

Note:

```
In case you need to disable the authentication feature, you have to use the root user:
$ etcdctl -u root auth disable
```

4.3 Working with ETCD

At this moment, we have our ETCD cluster completely configured. Therefore, it is time to start communicating with it. As we have explained, ETCD is a distributed databased based on a key-value store, so the information we need to obtain is saved in these keys, that are contained in directories. In this section we will see different manners to manage these keys, like its creation, tracking or updating.

There are two basic ways of interacting with ETCD: through the HTTP/JSON API and through a client. Again, we will use etcdctl, which is a command-line client for ETCD, and HTTP for those users that don't have the ETCD tools installed but want to use the app.

4.3.1 Keys management

First of all, we will use only our five servers, without considering external users. The cluster can read, write, track and remove keys from our system, even automatically (for configuration matters) as manually. Now, we have set several directories, but no keys. Let's create the first keys. From server1 terminal window, set

a key which name will be "shape" with an associated value "square":

```
server1$ etcdctl set /shape square
```

Now, add another key which name will be "color" and value "green". We will include another field in the order:

```
server1$ etcdctl set /color green -ttl 3
```

As we can observe the value of the keys as outputs, it is confirmed that have been created. Another server can read them, for example, server3. Let's go to this node and write:

```
server3$ etcdctl get /color
server3$ etcdctl get /shape
```

Take a look of the outputs:

- **Are both values shown?**
- **Which message appears? Why do you think is due to?**
- **What does "ttl" mean?**

Write again the key "color" in the ETCD, but now establish a TTL of 1000. Read that key one again:

- **Which is output now?**
- **What do you think it will happen after 1000 second?**

In the previous section, we have enabled the authentication feature. However, the orders below executed had not any field specifying which user will run the instruction. Therefore:

- **Is it the authentication feature working? Try to access to the ETCD config information, like the user list.**
- **Which user has been used for the previous keys management? Clue: look at the command windows where ETCD is running.**
- **Which are the permissions of that user? Do you think it is safe to keep this user with these credentials, based on our system?**

In order to keep a safe authentication system, we would revoke its permissions as follows:

```
$ etcdctl -u root role revoke  guest -path '/*' -readwrite
```

Now we have enabled definitively the authentication tool. To check it is working, execute the following order from one server:

```
 $ etcdctl ls
```

- **Can you see the list of directories? Which is the output?**

The user Admin1 wants to write a key in the directory "ETCD_information". To do so, he runs the following order:

```
$ etcdctl -u Admin1 set /ETCD_information/configAP 1
```

The user Client1 knows this key exists and wants to know its value, so he executes:

```
$ etcdctl -u Client1 get /ETCD_information/configAP
```

- **Which is the output?**
- **Can you know the value of "configAP"? Why?**

Clients have access to "Public_information" directory. Imagine Client2 and Admin1 want to write different keys in this directory:

```
$ etcdctl -u Client2 set /Public_information/logs 123
$ etcdctl -u Admin1 get /Public_information/logs
$ etcdctl -u Admin1 set /Public_information/term 12
```

- **Is there any problem? Why?**

However, for terms of simplicity we will maintain the guest role from now. Therefore, to recuperate its full permissions, execute:

```
$ etcdctl -u root role grant  guest -path '/*' -readwrite
```

Let's focus on the key "shape". Imagine that server4 is interested in its value, specifically, about when this value changes (maybe to execute an action when this happens). Therefore, this node will keep a tracking on the key, looking for an update.

In server4, execute the following order:

```
server4$ etcdctl watch /shape
```

Now, type this instruction in server1:

```
server1$ etcdctl set /shape circle
```

- **Which output is shown in server4?**

Imagine now that this node has the objective of keep a tracking of that key, without ending its operation when it changes. It would be interesting if the user just wants to have a record of the key updates. To achieve this aim, it is needed to add a new command option. In server4, execute:

```
server4$ etcdctl watch - -forever /shape
```

Set again new values for this key in server1 and check that the server4 shows the changes, but doesn't end as the previous case. This is because we have ordered that this node must watch the key forever. To finish its operation, just execute `CTRL+C`.

When a key is no longer needed and a TTL has not been set, there is an option to directly delete it. From server3, execute:

```
server4$ etcdctl rm /shape
```

- **Which output is shown in server3?**

Now imagine that an external node wants to start a communication with the ETCD cluster. This would be our client, which requests some information stored in our keys and also to write new ones. So we will start our LXC named "client", created at the beginning of this practice. The ETCD is not installed in this server, so we cannot use the "etcdctl" command. Now there is only one option: to use the HTTP-based API by means of the command cURL.

In a new terminal, execute the following lines:

```
$ lxc-start -n client
$ lxc-attach -n client
$ sudo apt-get install curl
```

Our client knows the IP address of server1, so it will send a request to this node in order to write a key named "shape" with value "square". In the client terminal, execute the following order (adapting the IP address):

```
 $ curl -L -X PUT http://10.0.3.194:2379/v2/keys/shape -d value="square"
```

From other server of the cluster, like server3, you can check if the key has been correctly set, also using the cURL command. In this case, as we are inside the ETCD cluster, it is not necessary to use a specific IP address of other node, but the localhost address:

```
$ curl  http://127.0.0.1:2379/v2/keys/shape
```

Now, take a look to both responses in client and server3 terminals after executing the previous orders.

- **Which is the difference between both outputs?**
- **Which is the equivalent orders using "etcdctl" command line?**

With the HTTP-based API it is also possible to establish a Time-To-Live period for the key expiration. For example, let's create a key name "color" with value "blue" and put a TTL of 500 seconds:

```
$ curl -L -X PUT http://10.0.3.194:2379/v2/keys/color?ttl=500 -d value="blue"
```

- **Which fields have been added? What do they mean?**

After executing this order, the user realize that the key must not be deleted after 500 seconds. In addition, the correct value associate to "color" must be "green", not "blue". In this case, we have to modified again its parameters:

```
$ curl -L -X PUT http://10.0.3.194:2379/v2/keys/color -d value="green" -d ttl=
  -d prevExist=true
```

- **Which is the name of the "action" field?**
- **How many fields are shown for the "node" attribute?**
- **Which is the value of the TTL shown in the prevNode attribute? And the indexes?**

- **What do you think the "prevExist" field stands for? What would it happen if we set its value in False?**
- **Create a new key named "number" and set its value to 5, adding the field "prevExist=True". Which is the output?**
- **Repeat again the last order, but adding "prevExist=False". Which is the value of the "action field"?**

Sometimes, a member could want to set several keys in a queue, without specifying a key name. It can be done by using the curl command with the action "POST". Let's execute in the client terminal:

```
$ curl -XPOST http://10.0.3.194:2379/v2/keys/queue -d value=FirstValue
$ curl -XPOST  http://10.0.3.194:2379/v2/keys/queue -d value=SecondValue
$ curl -XPOST  http://10.0.3.194:2379/v2/keys/queue -d value=ThirdValue
```

- **Which is the name of these keys?**

From server4, get the value of the queue in the creation order:

```
curl -s http://127.0.0.1:2379/v2/keys/queue?sorted=true
```

- **Is "/queue" a key?**
- **Which is the output if you don't add the field "sorted=true"?**

Once several keys have been creates, it is time to track some of them. First, from the client terminal, obtain the value of the key "shape". Thet output must be "square", as it has not been changed since we set it. Therefore, we will perform a tracking of this key. In the same terminal, execute the following instruction:

```
$ curl http://10.0.3.194:2379/v2/keys/shape?wait=true
```

Next, go to server3's terminal and assign to this key the value "triangle":

```
$ curl -L -X PUT http://127.0.0.1:2379/v2/keys/shape -d value="triangle"
```

- **Which is the output in the client terminal? Which action has been performed?**
- **Which is the ModifiedIndex?**

By executing the following command in a server of the cluster, it is possible to know all the information regarding a key:

```
$ curl 'http://127.0.0.1:2379/v2/keys/color' -vv
```

Looking at this data, identify the modifiedIndex and the createdIndex. In server1, run the following order, assigning to "waitIndex" the value of "createdIndex":

```
$ curl 'http://127.0.0.1:2379/v2/keys/color?wait=true&waitIndex='
```

- **Which is the action shown by the output?**

Now execute the same order, but assigning to "waitIndex" the value of "modifiedIndex":

- **Which is now the action shown?**
- **What is the purpose of these orders?**

Now, change the value of "waitIndex" for "modifiedIndex"+5:

- **Why is there not an output? Which is the order?**

From server4, change the value of the key "color" and assign the value "yellow" at the end:

- **Why is it not shown every change of the key in server1 terminal?**

We will end working with keys by removing them, as it has been seen for etcdctl command-line. The following command shows an example of how to delete a key. Execute it in client server:

```
$ curl -X DELETE http://10.0.3.194:2379/v2/keys/color?prevValue=grey
```

- **Has been the key deleted? Which is the output? Why do you think it is the reason, looking at the order and the message prompted?**
- **Execute a correct DELETE cURL order achieving a condition.**
- **How can this key deleted in general, regardless of its value?**

4.3.2 Directories management

Once we have seen ways to work with keys, we will proceed to do the same with directories. In the authentication section it already was shown a manner to create directories with the command-line etcdctl, and now we will try with the cURL command.

Execute the following order from server1:

```
$ curl http://127.0.0.1:2379/v2/keys/University/UPC -XPUT -d value=1
```

From client server, run this one:

```
$ curl http://10.0.3.194:2379/v2/keys/University/UPM -XPUT -d value=0
```

- **After running these instructions, what have we created? Identify every element.**
- **Is there any reference of a directory?**
- **Try to obtain the list of directories with etcdctl from server1. Can you differentiate the keys form the directories?**
- **Perform a GET operation from the client side with cURL to obtain the information of every key stored. Can you diferenciate now the keys from the directories?**

Let's create an empty Directory from server1, whose name will be "/_secret":

```
$ curl -X PUT http://127.0.0.1:2379/v2/keys/_secret  -d dir=true
```

As we can see, it has been set with the tag of "directory". Let's put now a key inside this directory from client server:

```
$ curl -L -X PUT http://10.0.3.194:2379/v2/keys/_secret/Conf -d value=yes
```

- **Run an order to list every directory as before, with cURL and etcdctl.**
- **Do you see the last directory we have created?**
- **Now, try to access directly to that directory by using a GET operation with cURL and etcdctl. Can you see it now? Why?**
- **Which should be the name of the directory to see it in the general list?**

Working with directories is pretty similar as with keys. Imagine we have to track the entire directory of "University" from client1.

- **Using the cURL command, which order would you run to watch every key inside it? How would it be using the command-line etcdctl?**
- **Using etcdctl, which instruction would you execute to track the keys inside "University" forever?**

Removing a directory is as easy as with keys: performing a DELETE operation. Run from the server1 the following order:

```
$ curl -X DELETE  'http://127.0.0.1:2379/v2/keys/University?dir=true'
```

- **Why does an error appears?**
- **In which conditions that directory would be deleted with that order?**
- **How have you to adapt the order to delete completely the directory "University"? Clue: you need to remove it recursively.**

Chapter 5

Conclusions

We have been able to verify the operation of the algorithms of consensus is very useful for its implementation in applications, since they provide a safe tool against failures of servers. Thanks to its performance, you get the same response from the cluster of nodes, regardless of who the request was made or the drop of a node involved. The information obtained will be consistent even if one server fails, since the rest must have agreed to provide a equal response to the client.

Based on consensus algorithms to give a secure and reliable background to your information (more specifically, in Raft), ETCD works as a database through which the cluster can easily share its different keys and records. Thanks to its user-friendly implementation, this system is simple and straightforward, allowing fast communication thanks to an accessible API through the cURL (HTTP) and etcdctl (command-line utility provided by ETCD) commands. In this way, the user can create, modify, delete and even track the keys that are in the system, as well as obtain information of operation of the different nodes of the cluster.

Even though it is advisable to configure the entire cluster before starting it (after having made an appropriate design of the system to be implemented), there is also a dynamic configuration setting. Thus, you can change the cluster size at any time, favoring scalability to a certain limit. It should be remembered that ETCD is oriented to storage of up to a few Gigabytes in a network of not many nodes, since latency is a critical factor in Raft (the system will be as slow as the slowest of nodes). However, this does not mean that it is ideal to have few servers communicating in the cluster because a good fault tolerance is also valued. For this same reason, we will always bet on an odd number of nodes, since the quorum would not change but would add an extra node to avoid falling into a general failure by server crashes.

Although in our case we have not used it, the Discovery service provided by ETCD offers the possibility of forming a new cluster using one already deployed. In case you do not have your own, you could use one of CoreOs for free, bearing in mind that it is only possible to access if you have a public IP. In our case, as we were working with LXC that only have private IPs, we have not been able to use it. However, having all the IPs of the nodes that would form the cluster, we did not have to, since we knew all the information necessary to bootstrapping the cluster in a static way.

We can provide the system with more security by implementing an authentication procedure based on write, read or total permissions assigned to specific roles. We have seen how they can control both individual keys and directories. Also that, by default there is a user named "guest" who has full system-wide permissions, and it would be necessary to revoke its privileges to limit access to unwanted users. However, as we have seen, only the authentication with the command-line etcdctl, not cURL, has been handled. This is not to say that it is not possible, but is in an experimental phase, so it has not been documented in this project.

We can conclude that ETCD is a very useful tool for its implementation in an endless number of applications that require a way to ensure and provide consistency of information without requiring complex queries or mass storage of logs. By sacrificing occasionally the availability of this information, it favors the reliability of its data to provide to the user always safe and consistent answers. As this project is in continuous development, new tools will be implemented, as well as existing ones will be perfectionated, offering in the end a database that without great pretensions, will help diverse systems to protect themselves and offer a safe and reliable service.

Annex A

Clustering parameters

A.0.1 Clustering Flags

In the following table, the different flags used for clustering configuration are described for both static and discovery service:

Flag	Description	Default
-initial-advertise-peer-urls	List of this member's peer URLs to advertise to the rest of the cluster. These addresses are used for communicating ETCD data around the cluster. At least one must be routable to all cluster members. These URLs can contain domain names.	"http://localhost:2380, http://localhost:7001"
-initial-cluster	Initial cluster configuration for bootstrapping.	"default=http://localhost:2380, default=http://localhost:7001"
-initial-cluster-state	Initial cluster state ("new" or "existing"). Set to new for all members present during initial static or DNS bootstrapping. If this option is set to existing, ETCD will attempt to join the existing cluster. If the wrong value is set, ETCD will attempt to start but fail safely.	"new"
-initial-cluster-token	Initial cluster token for the ETCD cluster during bootstrap.	"ETCD-cluster"
-advertise-client-urls	List of this member's client URLs to advertise to the rest of the cluster. These URLs can contain domain names.	"http://localhost:2379, http://localhost:4001"
-discovery	Discovery URL used to bootstrap the cluster.	None
-discovery-srv	DNS srv domain used to bootstrap the cluster.	None
-discovery-fallback	Expected behavior ("exit" or "proxy") when discovery services fails.	"proxy"
-discovery-proxy	HTTP proxy to use for traffic to discovery service.	None
-strict-reconfig-check	Reject reconfiguration requests taht would cause quorum loss.	False

Table A.1: Clustering flags

A.0.2 Clustering environmental variables

There is another manner to configure the cluster without using flags: working with environment variables. For the previously described flags, the following variables are used:

Flag	**Env. Variable**
-initial-advertise-peer-urls	ETCD_INITIAL_ADVERTISE_PEER_URLS
-initial-cluster	ETCD_INITIAL_CLUSTER
-initial-cluster-state	ETCD_INITIAL_CLUSTER_STATE
-initial-cluster-token	ETCD_INITIAL_CLUSTER_TOKEN
-advertise-client-urls	ETCD_ADVERTISE_CLIENT_URLS
-discovery	ETCD_DISCOVERY
-discovery-srv	ETCD_DISCOVERY_SRV
-discovery-fallback	ETCD_DISCOVERY_FALLBACK
-discovery-proxy	ETCD_DISCOVERY_PROXY
-strict-reconfig-check	ETCD_STRICT_RECONFIG_CHECK

Table A.2: Clustering environmental variables

Annex B

Member parameters

B.0.1 Member Flags

Once the cluster is configured and up, the next step is to take a look into the member's information, as well as the directory where shared data is stored. For performing a runtime reconfiguration the administrator has to include different flags that those seen in the previous subsection, as the cluster state has changed from "new" to "existing". These flags and several tunable parameters are described in the following table:

Flag	Description	Default
-name	Human-readable name for this member, indicated at the -initial-cluster flag. If the static bootstrapping is being used, it is necessary that this name matches that key in the flag. When using discovery, each member must have a unique name.	"default"
-data-dir	Path to the data directory.	"$name.ETCD"
-wal-dir	Path to the dedicated wal directory. If this flag is set, ETCD will write the WAL files to the walDir rather than the dataDir. This allows a dedicated disk to be used, and helps avoid IO competition between logging and other IO operations.	""
-snapshot-count	Number of committed transactions to trigger a snapshot to disk.	"10000"
-heartbeat-interval	Time (in milliseconds) of a heartbeat interval.	"100"
-election-timeout	Time (in milliseconds) for an election to timeout.	"1000"
-listen-peer-urls	List of URLs to listen on for peer traffic. This flag tells the ETCD to accept incoming requests from its peers on the specified scheme: //IP:port combinations. Scheme can be either http or https. If 0.0.0.0 is specified as the IP, ETCD listens to the given port on all interfaces. If an IP address is given as well as a port, ETCD will listen on the given port and interface. Multiple URLs may be used to specify a number of addresses and ports to listen on. The ETCD will respond to requests from any of the listed addresses and ports. For binding, domain name is invalid (for example, it is not correct "http://example.com:2380" for this flag, but "http://10.0.0.1:2380").	"http://localhost:2380, http://localhost:7001"
-listen-client-urls	List of URLs to listen on for client traffic. This flag tells the ETCD to accept incoming requests from the clients on the specified scheme: //IP:port combinations. Scheme can be either http or https. If 0.0.0.0 is specified as the IP, ETCD listens to the given port on all interfaces. If an IP address is given as well as a port, ETCD will listen on the given port and interface. Multiple URLs may be used to specify a number of addresses and ports to listen on. The ETCD will respond to requests from any of the listed addresses and ports. For binding, domain name is invalid (for example, it is not correct "http://example.com:2379" for this flag, but "http://10.0.0.1:2379").	"http://localhost:2379, http://localhost:4001"
-max-snapshots	Maximum number of snapshot files to retain (0 is unlimited).The default for users on Windows is unlimited, and manual purging down to 5 (or your preference for safety) is recommended.	"5"
-max-wals	Maximum number of wal files to retain (0 is unlimited). The default for users on Windows is unlimited, and manual purging down to 5 (or your preference for safety) is recommended.	"5"
-cors	Comma-separated white list of origins for CORS (cross-origin resource sharing).	None

Table B.1: Member flags

B.0.2 Member environmental variables

As mentioned in the previous section, it is possible to work with environmental variables instead of flags. The equivalence is shown in the following table:

Flag	**Env. Variable**
-name	ETCD_NAME
-data-dir	ETCD_DATA_DIR
-wal-dir	ETCD_WAL_DIR
-snapshot-count	ETCD_SNAPSHOT_COUNT
-heartbeat-interval	ETCD_HEARTBEAT_INTERVAL
-election-timeout	ETCD_ELECTION_TIMEOUT
-listen-peer-urls	ETCD_LISTEN_PEER_URLS
-listen-client-urls	ETCD_LISTEN_CLIENT_URLS
-max-snapshots	ETCD_MAX_SNAPSHOTS
-max-wals	ETCD_MAX_WALS
-cors	ETCD_CORS

Table B.2: Member environmental variables

Annex C

Libraries for ETCD

This annex documents the libraries for ETCD v.2 and its repositories in GitHub.

C.1 Go libraries

- etcd/client - the officially maintained Go client
- go-etcd - the deprecated official client. May be useful for older (<2.0.0) versions of etcd.

C.2 Java libraries

- boonproject/etcd - Supports v2, Async/Sync and waits
- justinsb/jetcd
- diwakergupta/jetcd - Supports v2
- jurmous/etcd4j - Supports v2, Async/Sync, waits and SSL
- AdoHe/etcd4j - Supports v2 (enhance for real production cluster)

C.3 Python libraries

- jplana/python-etcd - Supports v2
- russellhaering/txetcd - a Twisted Python library
- cholcombe973/autodock - A docker deployment automation tool
- lisael/aioetcd - (Python 3.4+) Asyncio coroutines client (Supports v2)

C.4 Node libraries

- stianeikeland/node-etcd - Supports v2 (w Coffeescript)
- lavagetto/nodejs-etcd - Supports v2
- deedubs/node-etcd-config - Supports v2

C.5 Ruby libraries

- iconara/etcd-rb
- jpfuentes2/etcd-ruby
- ranjib/etcd-ruby - Supports v2

C.6 C libraries

- jdarcy/etcd-api - Supports v2
- shafreeck/cetcd - Supports v2

C.7 C++ libraries

- edwardcapriolo/etcdcpp - Supports v2
- suryanathan/etcdcpp - Supports v2 (with waits)
- nokia/etcd-cpp-api - Supports v2
- nokia/etcd-cpp-apiv3

C.8 Clojure libraries

- aterreno/etcd-clojure
- dwwoelfel/cetcd - Supports v2
- rthomas/clj-etcd - Supports v2

C.9 Erlang libraries

- marshall-lee/etcd.erl

C.10 .Net Libraries

- wangjia184/etcdnet - Supports v2
- drusellers/etcetera

C.11 PHP Libraries

- linkorb/etcd-php

C.12 Haskell libraries

- wereHamster/etcd-hs

C.13 R libraries

- ropensci/etseed

C.14 Tcl libraries

- efrecon/etcd-tcl - Supports v2, except wait.

Bibliography

[1] Märt Bakhoff. *Consensus algorithms for distributed systems.*
`http://ds.cs.ut.ee/courses/course-files/MartBakhoff-consensus.pdf`

[2] Diego Ongaro and John Ousterhout. Stanford University. *Implementing Replicated Logs with Paxos.*
`https://ramcloud.stanford.edu/ ongaro/userstudy/paxos.pdf`

[3] Diego Ongaro and John Ousterhout. Stanford University. *Raft: A Consensus Algorithm for Replicated Logs.*
`https://raft.github.io/slides/raftuserstudy2013.pdf`

[4] Diego Ongaro and John Ousterhout. Stanford University. *In Search of an Understandable Consensus Algorithm (Extended Version).*
`https://raft.github.io/raft.pdf`

[5] CoreOS. *etcd | etcd Cluster.*
`https://coreos.com/etcd`

[6] CoreOS. *Distributed reliable key-value store for the most critical data of a distributed system.*
`https://github.com/coreos/etcd`

[7] Sreenivas Makam. *Service Discovery using etcd, Consul and Kubernetes.*
`https://es.slideshare.net/SreenivasMakam/service-discovery-using-etcd-consul-and-kubernetes`